Chapter 1

LU Factorization

1.1 Gauss elimination for dense matrix

1.1.1 Solution for triangular systems

Algorithm 1.1.1. To solve the triangular systems of equations $\mathbf{Ux} = \mathbf{c}$, where $\mathbf{U}$ is upper triangular, we use back-substitution as follows.

Algorithm 1: Back-substitution

```
Input: $\mathbf{U} \in \mathbb{R}^{n\times n}$, $\mathbf{c} \in \mathbb{R}^n$
Preconditions : $\mathbf{U}$ is upper triangular, $u_{kk} \neq 0$, $k = 1, 2, \dots, n$
Output: The solution $\mathbf{x}$ of $\mathbf{Ux} = \mathbf{c}$
1 $x_n = c_n/u_{nn}$;
2 for $k = n-1, n-2, \dots, 1$ do
3     $x_k = \left(c_k - \sum_{j=k+1}^{n} u_{kj}x_j\right)/u_{kk}$;
4 end
```

Algorithm 1.1.2. To solve the triangular systems of equations $\mathbf{Lx} = \mathbf{b}$, where $\mathbf{L}$ is lower triangular, we use forward substitution as follows.

Algorithm 2: Forward substitution

```
Input: $\mathbf{L} \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$
Preconditions : $\mathbf{L}$ is lower triangular, $l_{kk} \neq 0$, $k = 1, 2, \dots, n$
Output: The solution $\mathbf{c}$ of $\mathbf{Lc} = \mathbf{b}$
1 $c_1 = b_1/l_{11}$;
2 for $k = 2, 3, \dots, n$ do
3     $c_k = \left(b_k - \sum_{j=1}^{k-1} l_{kj}c_j\right)/l_{kk}$;
4 end
```

Proposition 1.1.3. Basic to all general-purpose direct methods for solving equation $\mathbf{Ax} = \mathbf{b}$ is the concept that triangular systems of equations are 'easy' to solve.

1.1.2 Gaussian transform

Definition 1.1.4 (Gaussian transform)**.** For a given vector $\mathbf{x} \in \mathbb{R}^n$, define

$$\mathbf{L}_k = \mathbf{I} - \mathbf{l}_k\mathbf{e}_k^T = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -l_{k+1,k} & 1 & & \\ & & \vdots & & \ddots & \\ & & -l_{nk} & & & 1 \end{bmatrix},$$

where $\mathbf{I}$ is the unitary matrix, $\mathbf{l}_k = (0, \dots, 0, l_{k+1,k}, \dots, l_{nk})^T$ and $l_{ik} = x_i/x_k, i = k+1, \dots, n$, $\mathbf{e}_k$ has an only nonzero entry 1 at the k-th component. Therefore, $\mathbf{L}_k\mathbf{x} = (x_1, \dots, x_k, 0, \dots 0)^T$. Matrix $\mathbf{L}_k$ is called *Gaussian transform or elementary lower triangular matrix.*

Lemma 1.1.5. $\mathbf{L}_k^{-1} = \mathbf{I} + \mathbf{l}_k\mathbf{e}_k^T$.

Proof. By definitions of $\mathbf{l}_k$ and $\mathbf{e}_k$, we have $\mathbf{e}_k^T\mathbf{l}_k = 0$, so

$$(\mathbf{I} - \mathbf{l}_k\mathbf{e}_k^T)(\mathbf{I} + \mathbf{l}_k\mathbf{e}_k^T) = \mathbf{I} - \mathbf{l}_k\mathbf{e}_k^T\mathbf{l}_k\mathbf{e}_k^T = \mathbf{I}.$$

□

Lemma 1.1.6.

$$\mathbf{L}_1^{-1} \dots \mathbf{L}_{n-1}^{-1} = \begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix}.$$

Proof. Notice that for $j < i$, $\mathbf{e}_j^T \mathbf{l}_i = 0$, so

$$\begin{aligned}&\mathbf{L}_1^{-1} \dots \mathbf{L}_{n-1}^{-1}\\ =&(\mathbf{I} + \mathbf{l}_1\mathbf{e}_1^T)(\mathbf{I} + \mathbf{l}_2\mathbf{e}_2^T)\dots(\mathbf{I} + \mathbf{l}_{n-1}\mathbf{e}_{n-1}^T)\\ =&\mathbf{I} + \mathbf{l}_1\mathbf{e}_1^T + \dots + \mathbf{l}_{n-1}\mathbf{e}_{n-1}^T\\ =&\begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix}.\end{aligned}$$

□

1.1.3 Gaussian elimination

Example 1.1.7. For the linear system

$$\mathbf{Ax} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix},$$

multiplying the first equation by $l_{21} = a_{21}/a_{11}$ (assuming $a_{11} \neq 0$) and subtracting from the second. Then, multiplying the first equation by $l_{31} = a_{31}/a_{11}$ and subtracting from the third. then we can get new linear systems

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2^{(2)} \\ b_3^{(2)} \end{bmatrix},$$

where $a_{ij}^{(2)} = a_{ij} - l_{i1}a_{1j}$, $b_i^{(2)} = b_i - l_{i1}b_1$, $i, j > 1$.

Finally, multiplying the new second row by $l_{32} = a_{32}^{(2)}/a_{22}^{(2)}$(assuming $a_{22}^{(2)} \neq 0$) and subtracting from the new third row produces the linear system

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & 0 & a_{33}^{(3)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2^{(2)} \\ b_3^{(3)} \end{bmatrix},$$

where $a_{33}^{(3)} = a_{33}^{(2)} - l_{32}a_{23}^{(2)}$, $b_3^{(3)} = b_3^{(2)} - l_{32}b_2^{(2)}$.

Notice that the above equation has the upper triangular form $\mathbf{Ux} = \mathbf{c}$ and we can simply use back-substitution to solve it.

Definition 1.1.8 (Gaussian elimination)**.** The procedure in Example 1.1.7 can be performed in general by creating zeros in the first columns, then the second and so forth. For $k = 1, 2, \dots, n-1$ we use the formulae

$$\begin{aligned} l_{ik} &= a_{ik}^{(k)}/a_{kk}^{(k)},\ i > k \\ a_{ij}^{(k+1)} &= a_{ij}^{(k)} - l_{ik}a_{kj}^{(k)},\ i, j > k \\ b_i^{(k+1)} &= b_i^{(k)} - l_{ik}b_k^{(k)},\ i > k, \end{aligned}$$

where $a_{ij}^{(1)} = a_{ij}$ and l_{ij} are called *multipliers.* This procedure is called *Gaussian elimination.* The only assumption required is that $a_{kk}^{(k)} \neq 0$, $k = 1, \dots, n$, these entries are called *pivots* in Gaussian elimination.

Lemma 1.1.9. By Gaussian elimination, we can factorize any $\mathbb{R}^{n\times n}$ matrix satisfied $a_{kk}^{(k)} \neq 0$, $k = 1, \dots, n$ as $\mathbf{A} = \mathbf{LU}$, where $\mathbf{L}$ is a unit lower triangular matrix and $\mathbf{U}$ is an upper triangular matrix. Thus, Gaussian elimination performs the same computation as LU factorization.

Proof. Notice that in each elimination step, the transform applied to the matrix is equivalent to a Gaussian transform $\mathbf{L}_k$. Define $\mathbf{A}^{(k)}$ as the matrix after $k-1$ elimination steps, then $\mathbf{A}^{(k+1)} = \mathbf{L}_k\mathbf{A}^{(k)}$. Using this relation for all values of k gives the equation

$$\mathbf{U} = \mathbf{A}^{(n)} = \mathbf{L}_{n-1} \dots \mathbf{L}_1\mathbf{A},$$

where $\mathbf{U}$ is an upper triangular matrix. Multiplying the above equation by $\mathbf{L}_1^{-1} \dots \mathbf{L}_{n-1}^{-1}$ on both sides gives the equation

$$\mathbf{A} = \mathbf{L}_1^{-1} \dots \mathbf{L}_{n-1}^{-1}\mathbf{U} = \mathbf{LU}.$$

By Lemma 1.1.6, we know that $\mathbf{L}$ is a unit lower triangular matrix. □

Algorithm 1.1.10. Observe that, in doing this computation on a computer, we may use a single two-dimensional array if we overwrite $\mathbf{A}^{(1)}$ by $\mathbf{A}^{(2)}$, $\mathbf{A}^{(3)}$, etc. Furthermore, each multiplier l_{ij} may overwrite the zero it creates. Thus, the array finally contains both $\mathbf{L}$ and $\mathbf{U}$ in packed form, excluding the unit diagonal of $\mathbf{L}$. The algorithm of Gaussian elimination is then as follows.

Algorithm 3: Gaussian elimination

```
Input: $\mathbf{A} \in \mathbb{R}^{n\times n}$
Postconditions: $\mathbf{L}$ is stored in the lower triangular part of $\mathbf{A}$ excluding the unit diagonal, $\mathbf{U}$ is stored in the upper triangular part of $\mathbf{A}$
1 for $k = 1, 2, \dots, n-1$ do
2     if $\mathbf{A}(k,k) == 0$ then
3         Gaussian elimination is failed
4     end
5     $\mathbf{A}((k+1):n, k) = \mathbf{A}((k+1):n, k)/\mathbf{A}(k,k)$;
6     $\mathbf{A}((k+1):n, (k+1):n) = \mathbf{A}((k+1):n, (k+1):n) - \mathbf{A}((k+1):n, k)\mathbf{A}(k, (k+1):n)$;
7 end
```

Theorem 1.1.11. The floating-point operations used in Gaussian elimination is $\frac{2}{3}n^3 + \mathcal{O}(n^2)$, where n is the dimension of the matrix.

Proof. Let r_k be the number of entries to the right of the main diagonal in row k of $\mathbf{A}^{(k)}$, let c_k be the number of entries below the main diagonal in column k of $\mathbf{A}^{(k)}$. For a dense matrix, these have values $r_1 = c_1 = n-1$, $r_2 = c_2 = n-2, \dots, r_n = c_n = 0$. Computing $\mathbf{A}^{(k+1)}$ from $\mathbf{A}^{(k)}$ invoves computing c_k multipliers and then performing $c_k r_k$ multiply-add pairs. The total cost of Gaussian elimination, excluding the work on the right-hand side, is

$$2\sum_{k=1}^{n-1} c_k r_k + \sum_{k=1}^{n-1} c_k = \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n.$$

□

Example 1.1.12 (Left-looking)**.** At each step in Gaussian elimination, the entire matrix below and to the right of the pivot was modified. For obvious reason, this is called *right-looking*. There are many alternatives, one is to delay the updates for a_{ij} until column j is about to be pivotal. Notice that

$$a_{ij} = \begin{cases} \sum_{1\le k\le i} l_{ik}u_{kj}, & i \le j, \\ \sum_{1\le k\le j} l_{ik}u_{kj}, & i \ge j. \end{cases}$$

When calculating the j-th column, we can first solve

$$\begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{j-1,1} & l_{j-1,2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{1j} \\ \vdots \\ u_{j-1,j} \end{bmatrix} = \begin{bmatrix} a_{1j} \\ \vdots \\ a_{j-1,j} \end{bmatrix}$$

to get $u_{1j}, \dots, u_{j-1,j}$. Then use

$$a_{ij}^{(j)} = a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj},\ i \ge j$$

to update $a_{ij}^{(j)}$. Finally, $l_{ij} = a_{ij}^{(j)}/a_{jj}^{(j)}$. This form is called *left-looking*. On modern hardware, it is usually faster because the intermediate values $a_{ij}^{(k)}, k = 2, \dots, j-1$ do not need to be stored temporarily in memory.

The difference between the left- and right-looking variants of LU factorization may be illustrated by considering which entries are active at each stage. Figure 1.1 shows what is stored at the begining of the third major processing step on a 5×5 matrix.

$$\begin{array}{ccccc} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ l_{21} & u_{22} & u_{23} & u_{24} & u_{25} \\ l_{31} & l_{32} & u_{33}^{\dagger} & u_{34}^{\dagger} & u_{35}^{\dagger} \\ l_{41} & l_{42} & a_{43}^{(3)*} & a_{44}^{(3)*} & a_{45}^{(3)*} \\ l_{51} & l_{52} & a_{53}^{(3)*} & a_{54}^{(3)*} & a_{55}^{(3)*} \end{array}$$

† Entries used. * Entries changed.

$$\begin{array}{ccccc} u_{11}^{\dagger} & u_{12}^{\dagger} & a_{13}^{*} & a_{14} & a_{15} \\ l_{21}^{\dagger} & u_{22}^{\dagger} & a_{23}^{*} & a_{24} & a_{25} \\ l_{31}^{\dagger} & l_{32}^{\dagger} & a_{33}^{*} & a_{34} & a_{35} \\ l_{41}^{\dagger} & l_{42}^{\dagger} & a_{43}^{*} & a_{44} & a_{45} \\ l_{51}^{\dagger} & l_{52}^{\dagger} & a_{53}^{*} & a_{54} & a_{55} \end{array}$$

† Entries used. * Entries changed.

Figure 1.1: Computational sequence for right- and left-looking LU factorization.

While the left-looking is more efficient on modern hardware than right-looking, it requires all the columns to the left of the active column to be accessed while it is being updated. Unless the matrix is small, these columns will not fit into the cache, so that access can be slow.

1.1.4 Partial pivoting

Example 1.1.13. The Gaussian elimination breaks down when $a_{kk}^{(k)} = 0$, illustrated by the case

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \end{bmatrix}.$$

Exchanging the first and second rows completely avoids this difficulty. So at the k-th step of Gaussian elimination where $a_{kk}^{(k)} = 0$, we have to exchange row k with row i satisfied $i > k$ and

$a_{ik}^{(k)} \neq 0$. The only way this can break down is if

$$a_{kk}^{(k)} = a_{k+1,k}^{(k)} = \ldots = a_{nk}^{(k)} = 0.$$

In this case, $\mathbf{A}$ is singular and the corresponding equation does not have a unique solution.

Example 1.1.14. Assuming we are solving

$$\begin{bmatrix} 0.001 & 2.42 \\ 1.00 & 1.58 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5.20 \\ 4.57 \end{bmatrix}$$

using a hypothetical computer with a 3-decimal floating-point representation of the form $\pm d_1.d_2d_3 \times 10^i$ where i is an integer, d_1, d_2, d_3 are decimal digits and $d_1 \neq 0$ unless $d_1 = d_2 = d_3 = 0$. The triangular system that results from applying Gaussian elimination is

$$\begin{bmatrix} 0.001 & 2.42 \\ 0 & -2420 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5.20 \\ -5200 \end{bmatrix}.$$

The computed solution is

$$\tilde{\mathbf{x}} = \begin{bmatrix} \tilde{x}_1 \\ \tilde{x}_2 \end{bmatrix} = \begin{bmatrix} 0.00 \\ 2.15 \end{bmatrix},$$

while the exact solution is

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \approx \begin{bmatrix} 1.18 \\ 2.15 \end{bmatrix}.$$

The error is not small compared with the exact solution.

However, when we exchange the first and the second row, the factorization will be

$$\begin{bmatrix} 1.00 & 1.58 \\ 0.001 & 2.42 \end{bmatrix} = \begin{bmatrix} 1 & \\ 0.001 & 1 \end{bmatrix} \begin{bmatrix} 1.00 & 1.58 \\ & 2.42 \end{bmatrix},$$

and the computed solution is

$$\tilde{\mathbf{x}} = \begin{bmatrix} \tilde{x}_1 \\ \tilde{x}_2 \end{bmatrix} \approx \begin{bmatrix} 1.17 \\ 2.15 \end{bmatrix},$$

which is almost correct.

Definition 1.1.15 (Partial pivoting)**.** One reason why we might get growth that destroys smaller values in the course of the computation comes from having a very large multiplier. A solution to this is to require that the inequality $|l_{ij}| \leq 1$ should hold for the coefficients of the matrix $\mathbf{L}$. This can be achieved by scanning the k-th column below the main diagonal to determine its entry of largest magnitude, the row containing this entry is exchanged with the k-th row, therefore $|a_{kk}^{(k)}| \geq |a_{ik}^{(k)}|$, $i > k$ now. This strategy is called *partial pivoting.*

Definition 1.1.16 (Full pivoting)**.** At each step in Gaussian elimination, we can choose the pivot to be the largest entry in the remaining submatrix rather than only in the k-th row. That is, row and column interchanges are performed at each step to ensure that $|a_{kk}^{(k)}| \geq |a_{ij}^{(k)}|$, $i \geq k, j \geq k$ all hold. This strategy is called *full* or *complete pivoting.*

Theorem 1.1.17. By Gaussian elimination with full pivoting, any nonsingular $\mathbb{R}^{n\times n}$ matrix can be factorized as $\mathbf{PAQ} = \mathbf{LU}$, where $\mathbf{P}$ and $\mathbf{Q}$ are permutation matrices, $\mathbf{L}$ is unit lower triangular and $\mathbf{U}$ is upper triangular.

Proof. Define elementary permutation matrix $\mathbf{I}_{pq}$ as the matrix obtained from $\mathbf{I}$ by exchanging the p-th and q-th columns (rows). $\mathbf{I}_{pq}\mathbf{A}$ exchanges the p-th and q-th rows of $\mathbf{A}$ and $\mathbf{AI}_{pq}$ exchanges the p-th and q-th columns of $\mathbf{A}$.

At each step of the full pivoting Gaussian elimination, the row interchange and column interchange can be represented as $\mathbf{P}_k = \mathbf{I}_{kp}$ and $\mathbf{Q}_k = \mathbf{I}_{kq}$, where (p, q) is the position of the largest entry. So the Gaussian elimination can be represented by

$$\mathbf{L}_{n-1}\mathbf{P}_{n-1}\ldots\mathbf{L}_1\mathbf{P}_1\mathbf{AQ}_1\ldots\mathbf{Q}_{n-1} = \mathbf{U}.$$

Define $\mathbf{Q} = \mathbf{Q}_1\ldots\mathbf{Q}_{n-1}$, $\mathbf{P} = \mathbf{P}_{n-1}\ldots\mathbf{P}_1$, $\mathbf{L} = \mathbf{P}(\mathbf{L}_{n-1}\mathbf{P}_{n-1}\ldots\mathbf{L}_1\mathbf{P}_1)^{-1}$, then we have

$$\mathbf{PAQ} = \mathbf{LU}.$$

It is obvious that $\mathbf{P}$ and $\mathbf{Q}$ are permutation matrices and $\mathbf{U}$ is upper triangular, so we only need to show $\mathbf{L}$ is unit lower triangular. Actually $\mathbf{L} = \mathbf{P}_{n-1}\ldots\mathbf{P}_2\mathbf{L}_1^{-1}\mathbf{P}_2\mathbf{L}_2^{-1}\ldots\mathbf{P}_{n-1}\mathbf{L}_{n-1}^{-1}$ because $\mathbf{P}_k\mathbf{P}_k = I$. Define

$\mathbf{L}^{(1)} = \mathbf{L}_1^{-1}$, $\mathbf{L}^{(k)} = \mathbf{P}_k\mathbf{L}^{(k-1)}\mathbf{P}_k\mathbf{L}_k^{-1}$, $k = 2, \ldots, n-1$,

then $\mathbf{L} = \mathbf{L}^{(n-1)}$. We can verify

$$\mathbf{L}^{(k)} = \begin{bmatrix} \mathbf{L}_{11}^{(k)} & 0 \\ \mathbf{L}_{21}^{(k)} & \mathbf{I}_{n-1} \end{bmatrix}, k = 1, \ldots, n-1 \qquad (1.1)$$

by induction, where $\mathbf{L}_{11}^{(k)}$ is a unit lower triangular matrix with entries norm no greater than 1, $\mathbf{L}_{21}^{(k)}$ is a $\mathbb{R}^{(n-k)\times k}$ matrix with entries norm no greater than 1, $\mathbf{I}_{n-1}$ is the unitary matrix of order $(n-k)$.

The induction base clearly holds for $k = 1$ because of the definition of $\mathbf{L}^{(1)} = \mathbf{L}_1^{-1}$. Now suppose that (1.1) holds for $k - 1$. Then

$$\mathbf{L}^{(k)} = \mathbf{P}_k\mathbf{L}^{(k-1)}\mathbf{P}_k\mathbf{L}_k^{-1} = \begin{bmatrix} \mathbf{L}_{11}^{(k-1)} & 0 \\ \tilde{\mathbf{L}}_{21}^{(k-1)} & \tilde{\mathbf{L}}_k^{-1} \end{bmatrix},$$

where $\tilde{\mathbf{L}}_{21}^{(k-1)}$ is obtained from $\mathbf{L}_{21}^{(k-1)}$ by exchanging its first and $(p-k+1)$-th rows, and

$$\tilde{\mathbf{L}}_k^{-1} = \begin{bmatrix} 1 & & & & \\ l_{k+1,k} & 1 & & & \\ l_{k+2,k} & 0 & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ l_{nk} & 0 & \cdots & 0 & 1 \end{bmatrix}.$$

So (1.1) holds for k, which complete the inductive proof. □

Proposition 1.1.18. Since partial pivoting can be viewed as a special case of full pivoting, any nonsingular $\mathbb{R}^{n\times n}$ matrix can be factorized as $\mathbf{PA} = \mathbf{LU}$, where $\mathbf{P}$ is a permutation matrix, $\mathbf{L}$ is unit lower triangular and $\mathbf{U}$ is upper triangular.

Algorithm 1.1.19. By Theorem 1.1.17, we can still use a single two-dimensional array by writing $\mathbf{L}$ in the lower triangular part excluding the unit diagonal. The full pivoting Gaussian elimination is then as follows.

Algorithm 4: Gaussian elimination with full pivoting

```
Input: $\mathbf{A} \in \mathbb{R}^{n\times n}$
Output: $\mathbf{u}, \mathbf{v} \in \mathbb{Z}^{n-1}$
Postconditions: $\mathbf{L}$ is stored in the lower triangular part of $\mathbf{A}$ excluding the unit diagonal, $\mathbf{U}$ is stored in the upper triangular part of $\mathbf{A}$, $\mathbf{u}, \mathbf{v}$ record $\mathbf{P}, \mathbf{Q}$ seprately
1  for $k = 1, 2, \dots, n-1$ do
2      Find $(p, q)$ s.t. $|\mathbf{A}(p,q)| = \max\{|\mathbf{A}(i,j)| : i = k : n, j = k : n\}$;
3      $\mathbf{A}(k, 1:n) \leftrightarrow \mathbf{A}(p, 1:n)$;
4      $\mathbf{A}(1:n, k) \leftrightarrow \mathbf{A}(1:n, q)$;
5      $u(k) = p$, $v(k) = q$;
6      if $\mathbf{A}(k,k) == 0$ then
7          Full pivoting Gaussian elimination is failed
8      end
9      $\mathbf{A}((k+1):n, k) = \mathbf{A}((k+1):n, k)/\mathbf{A}(k,k)$;
10     $\mathbf{A}((k+1):n, (k+1):n) = \mathbf{A}((k+1):n, (k+1):n) - \mathbf{A}((k+1):n, k)\mathbf{A}(k, (k+1):n)$;
11 end
```

1.1.5 Block factorization

Algorithm 1.1.20. In Example 1.1.12, we show that when the matrix is large, the columns in left-looking method will not fill into the cache, the solution to this problem is to group the updates by blocks. The most useful form is using the left-looking algorithm until a given number m of columns has been factorized, then performing a right-looking block update. If m columns fit into cache, we can factorize the first m columns with one data movement.

Let us partiation the matrix as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix},$$

where $\mathbf{A}_{11}$ is of order $m \times m$. Once the processing of the first m columns is complete, we will have the factorization

$$\begin{bmatrix} \mathbf{A}_{11} \\ \mathbf{A}_{21} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & \\ \mathbf{L}_{21} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} \\ 0 \end{bmatrix}.$$

If we had updated the remaining columns, the factorization would have been

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{11} & \\ \mathbf{L}_{21} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{11} & \mathbf{U}_{12} \\ & \ddot{\mathbf{A}}_{22} \end{bmatrix}.$$

By equating corresponding blocks we find the relations

$$\mathbf{L}_{11}\mathbf{U}_{12} = \mathbf{A}_{12},\ \ddot{\mathbf{A}}_{22} = \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{U}_{12},$$

$\ddot{\mathbf{A}}_{22}$ is known as the *Schur complement* of $\mathbf{A}_{11}$.

The columns of $\mathbf{U}_{12}$ may be found by forward substitution through $\mathbf{L}_{11}$. The remaining for the given matrix are exactly those that would be made for factorizing the matrix $\ddot{\mathbf{A}}_{22}$, this can be done by the same strategy.

Example 1.1.21 (Implicit block factorization)**.** In normal block factorization in Example 1.1.20, we calculate

$$\begin{aligned} &\mathbf{A}_{11} = \mathbf{L}_{11}\mathbf{U}_{11}, \\ &\mathbf{L}_{11}\mathbf{U}_{12} = \mathbf{A}_{12}, \\ &\mathbf{L}_{21}\mathbf{U}_{11} = \mathbf{A}_{21}, \\ &\ddot{\mathbf{A}}_{22} = \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{U}_{12} = \mathbf{L}_{22}\mathbf{U}_{22}. \end{aligned}$$

An interesting variation, known as *implicit block facotrization*, results if the factorization $\mathbf{A}_{11} = \mathbf{L}_{11}\mathbf{U}_{11}$ and $\ddot{\mathbf{A}}_{22} = \mathbf{L}_{22}\mathbf{U}_{22}$ are stored, but $\mathbf{U}_{12}$ is not. When $\mathbf{U}_{12}$ is needed as a multiplier, $\mathbf{L}_{11}^{-1}\mathbf{A}_{12}$

is used instead. This has little merit in the dense case, but in the sparse case it is extremely like that $\mathbf{U}_{12}$ has many more entries than $\mathbf{A}_{12}$ so less storage will be needed and sometimes less computation also.

1.2 Numerical consideration

Definition 1.2.1. To study the effect of arithmetic errors on the solution of sets of questions, we mainly need to answer two questions:

(i) Is the computed solution $\tilde{\mathbf{x}}$ the exact solution of a 'nearby' problem?

(ii) If small changes are made to the given problem, are changes in the exact solution also small?

A positive answer to the first question means the error in the computed solution is no greater than what would result from making small pertubations to the original problem and then solving the pertubed problem exactly. In this case, we say that the algorithm is *stable.*

If the answer to the second question is positive, we say that the problem is *well-conditioned.* Conversly, when small changes in the data produce large changes in the solution, we say the problem is *ill-conditioned.* This is a property of the problem and has nothing to do with the method used to solve it.

Definition 1.2.2. Define the *factorization-error matrix*

$$\mathbf{H} = \tilde{\mathbf{L}}\tilde{\mathbf{U}} - \mathbf{A},$$

where $\tilde{\mathbf{L}}\tilde{\mathbf{U}}$ is the numerical factorization of $\mathbf{A}$.

1.2.1 Controlling algorithm stability through pivoting

Example 1.2.3. In Example 1.1.14, we have showed that using Gaussian elimination without pivoting to solve

$$\begin{bmatrix} 0.001 & 2.42 \\ 1.00 & 1.58 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5.20 \\ 4.57 \end{bmatrix}$$

with our hypothetical 3-digit computer cause large error in the solution. Furthermore, the factorization-error matrix

$$\mathbf{H} = \begin{bmatrix} 0.0 & 0.0 \\ 0.0 & -1.58 \end{bmatrix}$$

is not small relative to $\mathbf{A}$. The residual is

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}} = \begin{bmatrix} -0.003 \\ 1.173 \end{bmatrix},$$

whose norm is not small compared with $\|\mathbf{b}\|$. So we conclude that our algorithm is unstable.

Note that the damage that lead to the inaccurate factorization is that it treats 1.58 as zero, the reason that it do so was the large growth in size that take place in forming $a_{22}^{(2)}$ from a_{22}.

Theorem 1.2.4. *Backward error analysis* shows that Gaussian elimination is stable provided the growth in Example 1.2.3 does not take place, actually we have the inequality

$$|h_{ij}| \leq 5.01\epsilon_u n \max_k |a_{ij}^{(k)}|,$$

where ϵ_u is the unit roundoff ($\epsilon_u = 0.005$ in our hypothetical 3-digit computer) and n is the dimenson of the matrix. However, it is not practical to control the sizes of all the coefficients $a_{ij}^{(k)}$, we usually weaken the inequality to the bound

$$|h_{ij}| \leq 5.01\epsilon_u n\rho, \tag{1.2}$$

where

$$\rho = \max_{i,j,k} |a_{ij}^{(k)}|.$$

Proof. See Reid [1971] combined with our wider bound on rounding error. □

Example 1.2.5 (Partial pivoting)**.** In practice, Gaussian elimination with partial pivoting is considered to be a stable algorithm. This is based on experience rather than rigorous analysis, since the best a priori bound that can be given for a dense matrix is

$$\rho \leq 2^{n-1} \max_{i,j} |a_{ij}|.$$

This bound is easy to establish, but generally very pessimistic, particularly in the case where n is very large.

Example 1.2.6 (Threshold pivoting)**.** While we want to control the size of the multiplier, we may find asking $|a_{kk}^{(k)}| \geq |a_{ik}^{(k)}|$, $i > k$ overly restrictive when other factors need to be taken into account, for example, sparsity or moving data in and out of cache. The compromise strategy is *threshold pivoting* which requires the inequality

$$|a_{kk}^{(k)}| \geq u|a_{ik}^{(k)}|,\ i > k,$$

where u, the *threshold parameter*, is a value in the range $0 < u \leq 1$. Partial pivoting can be regarded as a special threshold pivoting with $u = 1$. The growth bound of threshold pivoting is

$$\rho \leq (1 + u^{-1})^{n-1} \max_{i,j} |a_{i,j}|.$$

Example 1.2.7 (Rook pivoting)**.** Partial pivoting or threshold pivoting only limit the size of the multiplier, however, a small multiplier can produce a relatively large addition to an entry in the lower submatrix if an entry in the row is much larger than the pivot. *Rook pivoting* find the new pivot satisfied

$$|a_{kk}^{(k)}| \geq \max\{|a_{ik}^{(k)}|, |a_{ki}^{(k)}|\},\ i > k.$$

This limit the size both of the multiplier and the entry in the row. Foster [1997] has shown that, for rook pivoting, the growth is bounded by

$$\rho \leq 1.5 n^{3/4 \log(n)} \max_{i,j} |a_{i,j}|.$$

Example 1.2.8 (Full pivoting)**.** For full pivoting, Wilkinson [1961] has obtained that

$$\begin{aligned}\rho &\leq f(n) \max_{i,j} |a_{ij}| \\ &= \sqrt{n(2^1 3^{1/2} \ldots n^{1/(n-1)})} \max_{i,j} |a_{ij}|.\end{aligned}$$

Note that $f(n)$ is much more smaller than 2^{n-1} for large n.

Example 1.2.9 (SPD case)**.** When matrix $\mathbf{A}$ is SPD, things will be different. Wilkinson [1961] has shown that using Gaussian elimination without pivoting causes no growth i.e.

$$\rho = \max_{i,j,k} |a_{ij}^{(k)}| \leq \max_{i,j} |a_{ij}|.$$

So we do not need to consider pivoting when factorize a SPD matrix.

Example 1.2.10 (Diagonally dominant case)**.** When $\mathbf{A}$ is diagonally dominant, we can show that

$$\rho = \max_{i,j,k} |a_{i,j}^{(k)}| \leq 2 \max_{i,j} |a_{i,j}|.$$

(Hint: show that $\mathbf{A}^{(2)}$ is still diagonally dominant and $\sum_{j=2}^{n} |a_{ij}^{(2)}| \leq \sum_{j=1}^{n} |a_{ij}|$.)

1.2.2 Monitoring the stability

Definition 1.2.11. Since it is easy to compute $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$, and in practice this measure for the stability of the solution is usually employed. We mainly compare $\|\mathbf{r}\|$ with $\|\mathbf{b}\|$ or $\|\mathbf{A}\|\|\tilde{\mathbf{x}}\|$ to monitor the stability.

If $\|\mathbf{r}\|$ is small compared with $\|\mathbf{b}\|$, then we have solved a nearby problem since $\mathbf{A}\tilde{\mathbf{x}} = \mathbf{b} - \mathbf{r}$. By the first question in Definition 1.2.1, our method is stable.

If $\|\mathbf{r}\|$ is small compared with $\|\mathbf{A}\|\|\tilde{\mathbf{x}}\|$, then $\tilde{\mathbf{x}}$ is the exact solution of the equation $(\mathbf{A}+\mathbf{H})\tilde{\mathbf{x}} = \mathbf{b}$ where $\|\mathbf{H}\|$ is small compared with $\|\mathbf{A}\|$. By the first question in Definition 1.2.1, our method is stable.

Thus, if $\|\mathbf{r}\|$ is small compared with $\|\mathbf{b}\|$, $\|\mathbf{A}\tilde{\mathbf{x}}\|$, or $\|\mathbf{A}\|\|\tilde{\mathbf{x}}\|$, we have done a good job in solving the equation.

1.2.3 Ill-conditioning

Example 1.2.12. Suppose the matrix $\mathbf{A}$ is ill-conditioned, for example, there are two vector $\mathbf{v}$ and $\mathbf{w}$ satisfied

$$\|\mathbf{v}\| = \|\mathbf{w}\|,\ \|\mathbf{Av}\| \gg \|\mathbf{Aw}\|.$$

$\mathbf{v}$ is the solution of $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{b} = \mathbf{Av}$. Now we change $\mathbf{b}$ by the vector $\mathbf{Aw}$ which is relatively small but the corresponding solution changes a lot.

Clearly, the amount of ill-conditioning is dependent on $\|\mathbf{Av}\|/\|\mathbf{Aw}\|$, this ratio can be written in the form

$$\frac{\|\mathbf{Av}\|}{\|\mathbf{Aw}\|} = \frac{\|\mathbf{Av}\|}{\|\mathbf{Aw}\|}\frac{\|\mathbf{w}\|}{\|\mathbf{v}\|} = \frac{\|\mathbf{Av}\|}{\|\mathbf{v}\|}\frac{\|\mathbf{A}^{-1}\mathbf{y}\|}{\|\mathbf{y}\|},$$

where $\mathbf{y} = \mathbf{Aw}$. The right hand side has an upper bound $\|\mathbf{A}\|\|\mathbf{A}^{-1}\|$.

Definition 1.2.13 (Condition number)**.** The quantity

$$\kappa(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|$$

is known as the *condition number* of $\mathbf{A}$.

Theorem 1.2.14. For the general problem $\mathbf{Ax} = \mathbf{b}$, we consider its backward error. Assume $\mathbf{b}$ is perturbed by $\delta\mathbf{b}$ and the corresponding pertubation of $\mathbf{x}$ is $\delta\mathbf{x}$ so that equation

$$\mathbf{A}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}$$

holds. Then the relative error can be controlled by

$$\frac{\|\delta \mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa(\mathbf{A})\frac{\|\delta \mathbf{b}\|}{\|\mathbf{b}\|}.$$

Theorem 1.2.15. Now we consider the backward error for the matrix term. Let the perturbed equation be

$$(\mathbf{A}+\delta\mathbf{A})(\mathbf{x}+\delta\mathbf{x}) = \mathbf{b}.$$

Then the relative error can be controlled by

$$\frac{\|\delta \mathbf{x}\|}{\|\mathbf{x}+\delta\mathbf{x}\|} \leq \kappa(\mathbf{A})\frac{\|\delta \mathbf{A}\|}{\|\mathbf{A}\|}.$$

Theorem 1.2.16. When we consider the perturbation of $\mathbf{A}$ and $\mathbf{b}$ at the same time, we have the equation

$$(\mathbf{A}+\delta\mathbf{A})(\mathbf{x}+\delta\mathbf{x}) = \mathbf{b}+\delta\mathbf{b}.$$

When $\|\mathbf{A}^{-1}\|\|\delta\mathbf{A}\| < 1$, we have

$$\frac{\|\delta \mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\kappa(\mathbf{A})}{1-\kappa(\mathbf{A})\frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|}}\left[\frac{\|\delta \mathbf{b}\|}{\|\mathbf{b}\|}+\frac{\|\delta \mathbf{A}\|}{\|\mathbf{A}\|}\right].$$

Proof.

$$\begin{aligned}
&(\mathbf{A}+\delta\mathbf{A})(\mathbf{x}+\delta\mathbf{x}) = \mathbf{b}+\delta\mathbf{b},\\
\Rightarrow&\mathbf{A}\delta\mathbf{x} = \delta\mathbf{b}-\delta\mathbf{A}\mathbf{x}-\delta\mathbf{A}\delta\mathbf{x},\\
\Rightarrow&\|\delta\mathbf{x}\| \leq \|\mathbf{A}^{-1}\|(\|\delta\mathbf{b}\|+\|\delta\mathbf{A}\|\|\mathbf{x}\|+\|\delta\mathbf{A}\|\|\delta\mathbf{x}\|),\\
\Rightarrow&\|\delta\mathbf{x}\|(1-\|\mathbf{A}^{-1}\|\|\delta\mathbf{A}\|)\\
&\quad\leq \|\mathbf{A}^{-1}\|(\|\delta\mathbf{b}\|+\|\delta\mathbf{A}\|\|\mathbf{x}\|).
\end{aligned}$$

Provided the inequality $\|\mathbf{A}^{-1}\|\|\delta\mathbf{A}\| < 1$ holds, we deduce the relation

$$\|\delta\mathbf{x}\| \leq \frac{\|\mathbf{A}^{-1}\|}{1-\|\mathbf{A}^{-1}\|\|\delta\mathbf{A}\|}(\|\delta\mathbf{b}\|+\|\delta\mathbf{A}\|\|\mathbf{x}\|)$$

for the absolute error. For the relative error, we divide this by $\|\mathbf{x}\|$ and use the inequality $\|\mathbf{b}\| \leq \|\mathbf{A}\|\|\mathbf{x}\|$ to give the inequality

$$\frac{\|\delta \mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\kappa(\mathbf{A})}{1-\kappa(\mathbf{A})\frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|}}\left[\frac{\|\delta \mathbf{b}\|}{\|\mathbf{b}\|}+\frac{\|\delta \mathbf{A}\|}{\|\mathbf{A}\|}\right].$$

□

Definition 1.2.17 (Scaling)**.** *Scaling* is to select two diagonal matrices $\mathbf{D}_1$ and $\mathbf{D}_2$ such that $\mathbf{D}_1\mathbf{A}\mathbf{D}_2$ has a smaller condition number than $\mathbf{A}$.

Example 1.2.18. The condition number is very dependent on scaling. Consider the identity matrix with the last diagonal entry replaced by a small number δ, then $\kappa(\mathbf{A})$ will be around $1/\delta$. However, if the matrix is scaled to make its last entry 1, the condition number of the scaled matrix is just 1.

1.2.4 Strategies to improve the accuracy

Algorithm 1.2.19 (Iterative refinement)**.** Let $\mathbf{x}^{(1)}$ be the computed solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$, let the residual vector be

$$\mathbf{r}^{(1)} = \mathbf{b}-\mathbf{A}\mathbf{x}^{(1)}.$$

Formally, we observe the relations

$$\mathbf{A}^{-1}\mathbf{r}^{(1)} = \mathbf{A}^{-1}\mathbf{b}-\mathbf{x}^{(1)} = \mathbf{x}-\mathbf{x}^{(1)}.$$

Hence, we may find a correction $\Delta\mathbf{x}^{(1)}$ to $\mathbf{x}$ by solving

$$\mathbf{A}\Delta\mathbf{x}^{(1)} = \mathbf{r}^{(1)}.$$

In practive, of course, we can not solve this exactly, but we may construct a new approximation solution

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)}+\Delta\mathbf{x}^{(1)},$$

and, in general, define an iteration with steps

$$\begin{aligned}
&r^{(k)} = \mathbf{b}-\mathbf{A}\mathbf{x}^{(k)},\\
&\text{solve}\quad \mathbf{A}\Delta\mathbf{x}^{(k)} = \mathbf{r}^{(k)},\\
&\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}+\Delta\mathbf{x}^{(k)},
\end{aligned}$$

for $k = 1, 2, \ldots$.

Algorithm 1.2.20 (Automatic scaling)**.** A practical automatic scaling algorithm is to make all entries are close to 1. Let $\mathbf{D}_1 = \text{diag}(e^{-\rho_i})$ and $\mathbf{D}_2 = \text{diag}(e^{-\gamma_j})$, where ρ_i and γ_i, $i = 1, 2, \ldots, n$ are chosen to minimize

$$\sum_{a_{ij}\neq 0}(\log|a_{ij}|-\rho_i-\gamma_j)^2. \tag{1.3}$$

Then $\mathbf{D}_1\mathbf{A}\mathbf{D}_2$ has the logarithms of its nonzeros as small as possible in the least-squares sense. The minimization of (1.3) is a linear least-squares problem, which can be solved effectively by CG.

Algorithm 1.2.21 (Automatic scaling)**.** Another approach to scaling is to use an iterative technique to make the infinity norms of all the rows and columns close to one. Ruiz and Uçar [2014] acheive this by repeatedly scaling the rows and columns by the square roots of their norms. Starting with $\mathbf{A}^{\{0\}} = \mathbf{A}$ and

$$a_{ij}^{\{k+1\}} = \|a_{i:}^{\{k\}}\|_\infty^{-1/2}a_{ij}^{\{k\}}\|a_{:j}^{\{k\}}\|_\infty^{-1/2}.$$

Since any element of $\mathbf{A}^{\{k\}}$ is less than the norm of its row or column, all entries of $\mathbf{A}^{\{k\}}$, $k \geq 1$ are less than 1 in absolute value. If $a_{il}^{\{k\}}$ is the largest entry in row i, then

$$|a_{il}^{\{k+1\}}| = \|a_{i:}^{\{k\}}\|_\infty^{1/2} \|a_{:l}^{\{k\}}\|_\infty^{-1/2} \geq \|a_{i:}^{\{k\}}\|_\infty^{1/2}.$$

It follows that the norm of row i converge to 1 because

$$\|a_{i:}^{\{k+1\}}\|_\infty \geq \|a_{i:}^{\{k\}}\|_\infty^{1/2} \geq \|a_{i:}^{\{k\}}\|_\infty.$$

A similar results applies for the columns. Ruiz stops the iteration when

$$\max_{1\leq i\leq n} \left|1 - \|a_{i:}^{\{k\}}\|_\infty\right| \leq \epsilon \text{ and}$$
$$\max_{1\leq j\leq n} \left|1 - \|a_{:j}^{\{k\}}\|_\infty\right| \leq \epsilon.$$

Example 1.2.22. For the matrix

$$\mathbf{A} = \begin{bmatrix} 1.00 & 2420 \\ 1.00 & 1.58 \end{bmatrix},$$

the algorithm 1.2.20 yields the scaled matrix

$$\mathbf{D}_1\mathbf{A}\mathbf{D}_2 = \begin{bmatrix} 0.1598 & 6.2559 \\ 6.2559 & 0.1598 \end{bmatrix}.$$

The algorithm 1.2.21 yields the scaled matrix

$$\mathbf{D}_1\mathbf{A}\mathbf{D}_2 = \begin{bmatrix} 0.0228 & 1 \\ 1 & 0.0286 \end{bmatrix}.$$

1.3 Orderings in sparse Gaussian elimination

1.3.1 Orderings

Example 1.3.1. Consider the matrix that has the pattern illustrated on the right of Figure 1.2 with diagonal entries 1.0 and off-diagonal entries 1.1. The partial pivoting strategy would destroy all sparsity because none of the diagonal entries would be available as a pivot. However, if we adapt the threshold pivoting strategy with $u = 0.1$ to the sparse case, all the diagonal entries except the last are available as pivots and the sparsity is well preserved.

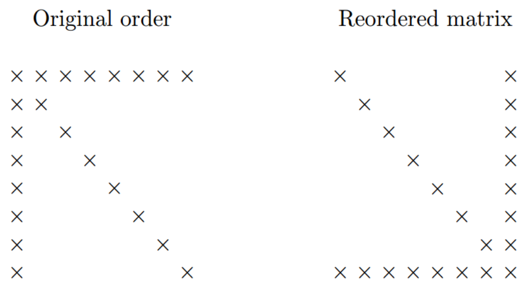


Figure 1.2: Reordering can preserve sparsity in factorization.

Remark 1.3.1. Ordering the rows and columns to preserve sparsity in Gaussian elimination was demonstrated to be effective in the Example 1.3.1. There are four quite different strategies for ordering a sparse matrix which are introduced in the following section:

1. We may be able to reorder the matrix to block triangular form. We discuss this in Section 1.3.2. If such an ordering exists, we can confine the factorization steps to smaller blocks on the diagonal, rather than to the whole matrix.

2. We consider local strategies where, at each stage of the factorization, we choose as pivot the entry that preserves sparsity "as well as possible" according to some criteria, which will be introduced in Section 1.3.3.

3. There are methods of preserving sparsity by confining fill-ins to particular areas of the matirx, which will be developed in Section 1.3.4.

4. A strategy under the heading of dissection methods looks at the matrix in its entirety, seeking to split the overall problem into independent subproblems, which is particularly applicable for parallel computing environments. We will discuss in Section 1.3.5.

There is no single "best" method for preserving sparsity. The criteria for what is best is more difficult than it might seem at first. For example, the criterion of "least fill-in" might seem to be best, but may require a great deal more data structure manipulation.

1.3.2 Reduction to block triangular form

Definition 1.3.2. A pattern is worthwhile to seek is the *block lower triangular form*

$$\mathbf{PAQ} = \begin{bmatrix} \mathbf{B}_{11} & & & & \\ \mathbf{B}_{21} & \mathbf{B}_{22} & & & \\ \mathbf{B}_{31} & \mathbf{B}_{32} & \mathbf{B}_{33} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \mathbf{B}_{N1} & \mathbf{B}_{N2} & \mathbf{B}_{N3} & \cdots & \mathbf{B}_{NN} \end{bmatrix} \tag{1.4}$$

Definition 1.3.3. A matrix that can be permuted to the form (1.4), with $N > 1$, is said to be *reducible*. If no block triangular form other than the trivial one can be found, the matrix is called *irreducible*. We expect each $\mathbf{B}_{ii}$ to be irreducible, for otherwise a finer decomposition is possible.

Algorithm 1.3.4. If we partition $\mathbf{x}$ and $\mathbf{b}$ similarly, we may solve the equation $\mathbf{Ax} = \mathbf{b}$ by solving the simple forward substitution.

$$\mathbf{B}_{ii}\mathbf{y}_i = (\mathbf{Pb})_i - \sum_{j=1}^{i-1} \mathbf{B}_{ij}\mathbf{y}_j, \; i = 1, 2, \ldots, N \tag{1.5}$$

and the permutation

$$\mathbf{x} = \mathbf{Qy}. \tag{1.6}$$

Remark 1.3.2. We have to factorize only the diagonal blocks $\mathbf{B}_{ii}$. The off-diagonal blocks $\mathbf{B}_{ij}$, $i > j$, are used only in the multiplications $\mathbf{B}_{ij}\mathbf{y}_j$. In particular, all fill-in is confined to the blocks on the diagonal. Any row and column interchanges needed for the sake of stability and sparsity may be performed within the blocks on the diagonal and do not affect the block triangular structure.

Algorithm 1.3.5. We may find the block triangular form in three stages:

(i) Look for row and column singletons.

(ii) Permute entries onto the diagonal (usually called finding a *transversal*).

(iii) Use symmetric permutations to find the block form itself.

In this section, we discuss the three stages separately.

Algorithm 1.3.6. When we look for row and column singletons,

(i) If the matrix has a row singleton and we permute it to the position $(1, 1)$, the matrix has the form

$$\mathbf{PAQ} = \begin{bmatrix} \mathbf{B}_{11} & \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix}$$

where $\mathbf{B}_{11}$ has order 1. If the matrix $\mathbf{B}_{22}$has a row singleton, we may permute it to the position $(2, 2)$. Continuing in this way until no row singletons are available, we find the permuted form with $\mathbf{B}_{11}$ a lower triangular matrix.

(ii) Similarly, we may look for column singletons successively and permute each in turn to the end of the matrix. Following this we have the form

$$\mathbf{PAQ} = \begin{bmatrix} \mathbf{B}_{11} & & \\ \mathbf{B}_{21} & \mathbf{B}_{22} & \\ \mathbf{B}_{31} & \mathbf{B}_{32} & \mathbf{B}_{33} \end{bmatrix}$$

where $\mathbf{B}_{11}$ and $\mathbf{B}_{33}$ are lower triangular matices.

Example 1.3.7. We show an example in Figure 1.3. There is one row singleton and choosing it creates another. There are two column singletons and choosing them creates another. We are left with a middle block of size 2.

$$\begin{array}{c|ccccccc|} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & \times & & & \times & & & \\ 2 & & \times & & \times & & \times & \\ 3 & & \times & \times & & & & \times \\ 4 & \times & & & \times & & & \\ 5 & & \times & & & \times & & \\ 6 & & & & & & & \times \\ 7 & & & & & & \times & \times \end{array} \quad \text{becomes} \quad \begin{array}{c|ccccccc|} & 7 & 6 & 1 & 4 & 2 & 3 & 5 \\ \hline 6 & \times & & & & & & \\ 7 & \times & \times & & & & & \\ 1 & & & \times & \times & & & \\ 4 & & & \times & \times & & & \\ 2 & & \times & & \times & \times & & \\ 3 & \times & & & & \times & \times & \\ 5 & & & & & \times & & \times \end{array}$$

Figure 1.3: Permuting row and column singletons.

Remark 1.3.3. In the rest of the section, we consider how $\mathbf{B}_{22}$ can be permuted to block triangular form.

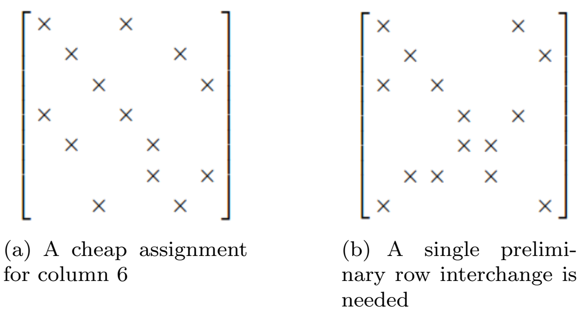
Definition 1.3.8. The algorithm of finding a transversal can be described in either a row or a column orientation. We follow the variant that looks at the columns one by one and permutes the rows. After permutations have been found that place entries in the first $k - 1$ diagonal positions, we examine column k and seek a row permutation that will:

(i) Preserve the presence of entries in the first $k - 1$ diagonal positions,

(ii) result in column k having an entry in row k.

The algorithm continues in this fashion extending the transversal by one at each stage. Success in this extension is called an *assignment*. Sometimes this transversal extension step is trivial, which is called a *cheap assignment*.

Example 1.3.9. In Figure 1.4(a), the assignment in column six is made by the interchange of rows six and seven. Figure 1.4(b) shows a simple case where a cheap assignment is not available, but the interchange of rows one and seven is adequate the $(1, 1)$ entry and make a cheap assignment possible in column six.


(a) A cheap assignment for column 6

(b) A single preliminary row interchange is needed

Figure 1.4: Two examples of assignment.

Definition 1.3.10. Sometimes the transversal cannot be extended because the matrix is singular for all numerical values of the entries. Such a matrix is called *symbolically singular* or *structurally singular*.

Example 1.3.11. A symbolically singular case is

$$\begin{bmatrix} \times & \times & & \times & \\ & & \times & & \times \\ & & \times & & \times \\ \times & \times & & \times & \\ & & \times & & \times \end{bmatrix}.$$

Algorithm 1.3.12 (Transversal extension by depth-first search)**.** We seek a sequence of columns $c_1, c_2, \ldots, c_j$ with $c_1 = k$ having entries in rows $r_1, r_2, \ldots, r_j$, with $r_i = c_{i+1}$, $i = 1, 2, \ldots, j-1$ and $r_j \geq k$. Then the sequence of row interchanges $(r_1, r_2), \ldots, (r_{j-1}, r_j)$ achieves what we need. To find this we start with the first entry in column k we take its row number to indicate the next column and continue letting the first off-diagonal entry in each column indicate the subsequent column. In each column, we look for an entry in row k or beyond. This sequence has one of the following possible outcomes:

(i) We find a column with an entry in row k or beyond.

(ii) We reach a row already considered.

(iii) We come to a dead end (that is a column with no off-diagonal entries or one whose off-diagonal entries have all already been considered).

In case (i) we have the sequence of columns that we need. In case (ii) we take the next entry in the current column. In case (iii), we backtrack to the previous column and start again with the next entry there.

Example 1.3.13. In figure 1.4(b), we seek a sequence of columns $6, 1, 7$ by the above algorithm, so the interchange of rows 1 and 7 is adequate to preserve the $(1, 1)$ entry and make a cheap assignment possible in column 6.

Remark 1.3.4. Always taking the next column rather than trying other rows in the present column is the depth-first part. Looking to see if there is an entry in row k or beyond is the look-ahead feature.

Theorem 1.3.14. If the matrix order is n and it has τ entries, the number of elementary operations of algorithm 1.3.12 is at worst proportional to $n\tau$.

Remark 1.3.5. We assume that a row permutation $\mathbf{P}_1$ has been computed so that $\mathbf{P_{1A}}$ has entries on every position on its diagonal, then we wish to find a permutation matrix $\mathbf{Q}$ such that $\mathbf{Q}^T(\mathbf{P}_1\mathbf{A})\mathbf{Q}$ has the form

$$\mathbf{Q^T}(\mathbf{P}_1\mathbf{A})\mathbf{Q} = \begin{bmatrix} \mathbf{B}_{11} & & & & \\ \mathbf{B}_{21} & \mathbf{B}_{22} & & & \\ \mathbf{B}_{31} & \mathbf{B}_{32} & \mathbf{B}_{33} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \mathbf{B}_{N1} & \mathbf{B}_{N2} & \mathbf{B}_{N3} & \cdots & \mathbf{B}_{NN} \end{bmatrix} \tag{1.7}$$

Remark 1.3.6. It is convenient to describe algorithms for this process with the help of the directed graphs associated with the matrices. Applying a symmetric permutation to the matrix causes no change in the associated directed graph except for the relabelling of its nodes.

Theorem 1.3.15. If we cannot find a closed path, we must be able to divide the directed graph into two parts such that there is no path from the first part to the second. Renumbering the first group of nodes $1, 2, \ldots, k$ and the seconde group $k+1, \ldots, n$ will produce a corresponding matrix in block lower triangular form.

Definition 1.3.16. The same process in theorem 1.3.15 may be applied to each resulting block until no fuether subdivision is possible. The sets of nodes corresponding to the resulting diagonal blocks are called *strong components*.

Example 1.3.17. In Figure 1.5, there is no connection from nodes $(1, 2)$ to nodes $(3, 4, 5)$. The corresponding matrix is of block lower triangular form.

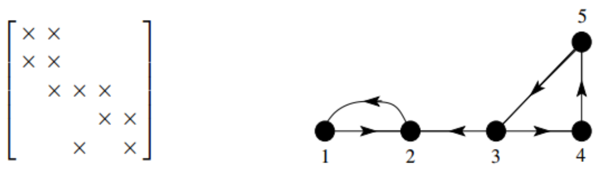


Figure 1.5: A 5×5 matrix and its digraph(directed graph).

Theorem 1.3.18. If $\mathbf{A}$ is a symmetric permutation of a triangular matrix, there must be a node in its digraph from which no path leaves.

Algorithm 1.3.19. The algorithm for finding the triangular form may be built upon the theorem 1.3.18:

1. We may start anywhere in the digraph and trace a path until we encounter a node from which no paths leave,

2. number the node at the end of the path first, and remove it and all edges pointing to it from the digraph,

3. continue from the previous node on the path until once again we reach a node with no path leaving it.

Example 1.3.20. We illustrate the algorithm with the digraph of Figure 1.6. The sequence of paths is illustrated in Figure 1.7.

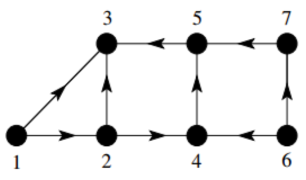


Figure 1.6: A digraph corresponding to a triangular matrix

Path					**5**						
			3	4	4	**4**					
		2	2	2	2	2	**2**			**7**	
	1	1	1	1	1	1	1	**1**	6	6	**6**
Step	1	2	3	4	5	6	7	8	9	10	11

Figure 1.7: The sequence of paths used for the Figure 1.6 case, where nodes selected for ordering are shown in bold.

Definition 1.3.21. *A composite node* denotes any group of nodes through which a closed path has been found. Edges within a composite node are ignored, and edges entering or leaving any node of the composite node are regarded as entering or leaving the composite node. It generalized the idea of algorithm 1.3.19 to the block case.

Algorithm 1.3.22 (The algorithm of Sargent and Westerberg)**.** Starting from any node, a path is followed through the digraph until:

1. A closed path is found (identified by encountering the same node or composite node tweice), or

2. a node or composite node is encountered with no edges leaving it.

In case (i), all the nodes on the closed path must belong to the same strong component and the digraph is modified by collapsing all nodes on the closed path into a single composite node. The path is now continued from the composite node. In case (ii), as for ordinary nodes in the triangular case, the composite node is numbered next in the relabelling. It and all edges connected to it are removed, and the path now ends at the previous node or composite node, or starts from any remaining node if it would otherwise be empty.

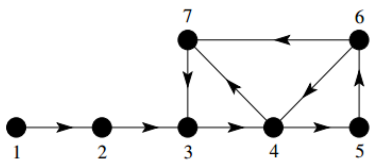


Figure 1.8: A digraph illustrating the algorithm of Sargent and Westerberg.

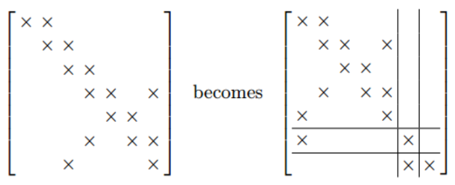


Figure 1.9: The matrices before and after renumbering.

Example 1.3.23. We illustrate with the example shown in Figure 1.9. Starting the path at node 1, it continues $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 4$, then $(4, 5, 6)$ is recognized as a closed path, and nodes 4,5,6 are relabelled as composite node $4'$. The path is continued from this composite node to become $1 \to 2 \to 3 \to 4' \to 7 \to 3$. Again, a closed path has been found and $(3, 4', 7)$ is labelled as $3'$ and the path becomes $1 \to 2 \to 3'$. Since there are no edges leaving $3'$, it is numbered first and removed. The path is now $1 \to 2$ and no edges leave node 2, so this is numbered second. Finally, node 1 is numbered as the last block. The corresponding original and reordered matrices are shown in Figure 1.9.

Remark 1.3.7. The difficulty with this approach is that there may be large overheads associated with the relabelling in the node collapsing step. A simple scheme such as labelling each composite node with the lowest label of its constituent nodes can result in $\mathcal{O}(n^2)$ relabellings.

Example 1.3.24. In Figure successive composite nodes are (4, 5), (3, 4, 5, 6), (2, 3, 4, 5, 6, 7), (1, 2, 3, 4, 5, 6, 7, 8). In general, such a digraph with n nodes will involve $2+4+6+\cdots+n = n^2/4+n/2$ relabellings.

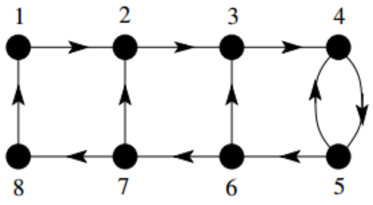


Figure 1.10: A case causing many relabellings.

1.3.3 Local pivotal strategies for sparse matrices

Definition 1.3.25. Suppose Gaussian elimination applied to an $n \times n$ matrix has proceeded through the first $k-1$ stages. For each row i in the active $(n-k+1) \times (n-k+1)$ submatrix, let $r_i^{(k)}$ denotes the number of entries in row i. Similarly, let $c_j^{(k)}$ be the number of entries in column j. Then, the Markowitz criterion is to select the entry $a_{ij}^{(k)}$ that minimizes the expression

$$(r_i^{(k)} - 1)(c_j^{(k)} - 1) \tag{1.8}$$

from the entries of the active $(n-k+1)\times(n-k+1)$ submatrix that are not too small numerically.

Remark 1.3.8. An advantage of using (1.8) rather than $r_i^{(k)}c_j^{(k)}$ is that this forces the algorithm to select a row singleton or a column singleton if either is present. Such a choice produces no fill-in at all.

Theorem 1.3.26. If $\mathbf{A}$ is symmetric, the search for the pivot is simplified to finding i such that

$$r_i^{(k)} = \min_t r_t^{(k)}$$

leading to $a_{ii}^{(k)}$ as pivot if $a_{ii}^{(k)}$ is nonzero.

Remark 1.3.9. It is called the minimum degree algorithm because of its graph theoretic interpretation: in the graph assoiated with a symmetric sparse matrix, this strategy corresponds to choosing the node for the next elimination that has the least edges connected to it.

Algorithm 1.3.27. Assoicated with the Markowitz ordering strategy is the need in the unsymmetric case to establish a suitable threshold parameter, u, for numerical stability. In particular, we restrict the Markowitz selection to those pivot candidates satisfying the inequality

$$|a_{kk}^{(k)}| \geq u|a_{ik}^{(k)}|, \ i > k, \tag{1.9}$$

where u is a preset parameter in the range $0 < u \leq 1$.

1.3.4 Ordering sparse mtrices for band solution

Remark 1.3.10. We consider a different approach to reordering to preserve sparsity, with algorithms that take a global view of the problem.

Definition 1.3.28. A sparse matrix in which the nonzero elements are located in a band about the main diagonal is called a *band matrix*. If $\mathbf{A}$ is a band matrix, then

$$a_{ij} = 0, \ |i-j| > s \text{ for some value } s,$$

which is illustrated on the left of Figure 1.11.

Definition 1.3.29. For a symmetrically structured matirx $\mathbf{A}$ has *bandwidth* $2m+1$ and *semibandwidth* m if m is the smallest integer such that $a_{ij} = 0$ whenever $|i-j| > m$. For unsymmetric case, we define the lower (upper) semibandwidth as the smallest integer $m_l(m_u)$ such that if a_{ij} is an entry, $i-j \leq m_l$ $(j-i \leq m_u)$. The bandwidth is $m_l + m_u + 1$.

Definition 1.3.30. $\mathbf{A}$ is of the *variable-band* form (also called skyline form) if

$$a_{ij} = 0,\ j - i > s_i \text{ or } a_{ji} = 0,\ j - i > t_i$$

for some values s_i and t_i, $i = 1, 2, \ldots, n$, which is shown on the right of Figure 1.11.

Definition 1.3.31. When using the variable-band form for a symmetric matrix, we store for each row every coefficient between the first entry in the row and the diagonal. In the unsymmetric case, we also store for each column every coefficient between the first entry in the column and the diagonal. The total number of coefficients stored is called the *profile*.

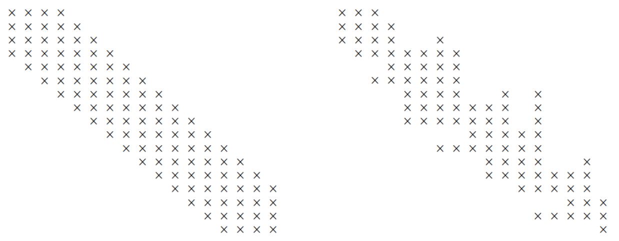


Figure 1.11: Band and variable-band matrices.

Remark 1.3.11. Finding the banded or variable-band form is based on finding permutations to block tridiagonal form. If the blocks are small and numerous such a matrix is banded with a small bandwidth.

Algorithm 1.3.32. The factorization of a block tridiagonal matrix A can be written in the form

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & & & \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} & & \\ & \mathbf{A}_{32} & \mathbf{A}_{33} & \cdot & \\ & & \cdot & \cdot & \cdot \\ & & & \cdot & \cdot \\ & & & & \cdot \quad \mathbf{A}_{NN} \end{bmatrix} = \mathbf{LU},$$

$\mathbf{L}$ and $\mathbf{U}$ can be written in

$$\mathbf{L} = \begin{bmatrix} \mathbf{I} & & & \\ \mathbf{A}_{21}\mathbf{D}_1^{-1} & \mathbf{I} & & \\ & \ddots & \ddots & \\ & & \mathbf{A}_{N,N-1}\mathbf{D}_{N-1}^{-1} & \mathbf{I} \end{bmatrix},$$

$$\mathbf{U} = \begin{bmatrix} \mathbf{D}_1 & \mathbf{A}_{12} & & & \\ & \mathbf{D}_2 & \mathbf{A}_{23} & & \\ & & \ddots & \ddots & \\ & & & \ddots & \mathbf{A}_{N-1,N} \\ & & & & \mathbf{D}_N \end{bmatrix},$$

where

$$\mathbf{D}_1 = \mathbf{A}_{11}, \tag{1.10}$$

$$\mathbf{D}_i = \mathbf{A}_{ii} - \mathbf{A}_{i,i-1}\mathbf{D}_{i-1}^{-1}\mathbf{A}_{i-1,i},\ i = 2, \ldots, N. \tag{1.11}$$

To use this form to solve a set of equations $\mathbf{Ax} = \mathbf{b}$ requires the forward substitution steps

$$\mathbf{c}_1 = \mathbf{b}_1 \tag{1.12}$$

$$\mathbf{c}_i = \mathbf{b}_i - \mathbf{A}_{i,i-1}\mathbf{D}_{i-1}^{-1}\mathbf{c}_{i-1},\ i = 2, \ldots, N. \tag{1.13}$$

followed by the substitution steps

$$\mathbf{x}_N = \mathbf{D}_N^{-1}\mathbf{N}\mathbf{c}_N, \tag{1.14}$$

$$\mathbf{x}_i = \mathbf{D}_i^{-1}\mathbf{c}_i - \mathbf{A}_{i,i+1}\mathbf{c}_{i+1},\ i = N - 1, \ldots, 1. \tag{1.15}$$

Remark 1.3.12. We hope to find an automatic ordering algorithm, such algorithms for the symmetric case are the subject of next section.

Algorithm 1.3.33 (Cuthill-McKee algorithm)**.** Symmetric permutations of a symmetric matrix correspond to relabellings of the nodes of the associated graph and it is easier to describe algorithms in terms of relabelling graphs. The following is the steps of Cuthill-McKee algorithm:

(i) Divide the nodes into *level sets*, S_i, with S_1 consisting of a single node. The next, S_2, consists of all the neighbours of this node. The set S_3 consists of all the neighbours of the nodes in S_2 that are not in S_1 or S_2. The general set S_i consists of lal the neighbours of the nods of S_{i-1} that are not in S_{i-2} and S_{i-1}.

(ii) Order within each block S_i by taking first those nodes that are neighbours of the first node in S_{i-1}, then those that are neighbours of the second node in S_{i-1}, and so on.

Example 1.3.34. The ordering of the Figure 1.12 could have come from level sets (1), (2-4), (5-7), (8-10), (11-13), (14-16), (17) and the corresponding matrix shown in Figure 1.13, is block tridiagonal with diagonal blocks having orders 1, 3, 3, 3, 3, 3 and 1.

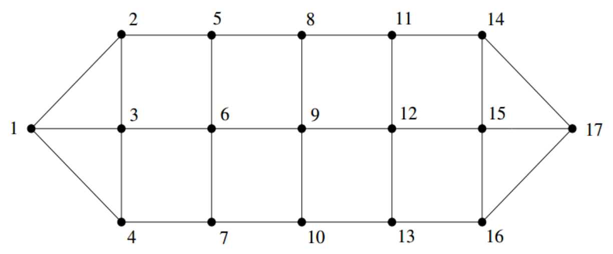


Figure 1.12: A graph with an obviously good node order.

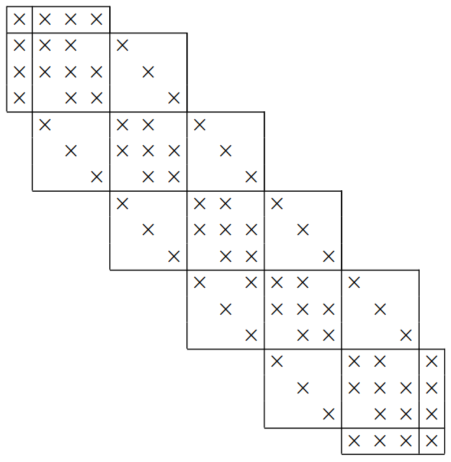


Figure 1.13: The matrix corresponding to the Figure 1.12 graph.

Algorithm 1.3.35 (the Reverse Cuthill-McKee algorithm(RCM))**.** The only difference with the Cuthill-McKee algorithm is that the reverse Cuthill-McKee algorithm orders from the last level set to S_1.

Remark 1.3.13. Reversing the Cuthill-McKee order often yields a worthwhile improvement, not in the bandwidth, but in the total storage required within the variable-band form (the profile) and in the number of arithmetic operations required for variable-band factorization.

Example 1.3.36. the Cuthill-McKee order (with level sets (1), (2), (3-7)) is shown in Figure 1.14 and there are 20 zeros within the variable-band form. On reversing the order (Figure 1.15) all these zeros move outside the form and will not need storage.

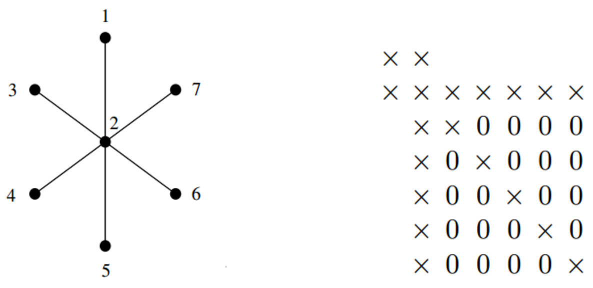


Figure 1.14: A graph and its associated matrix, ordered by Cuthill-McKee.

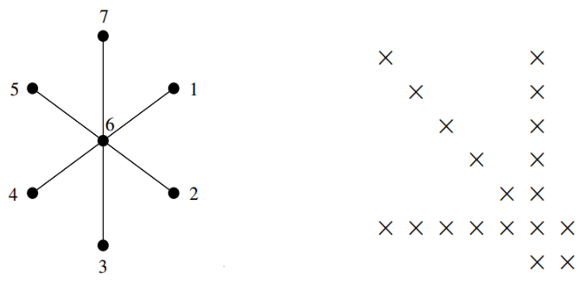


Figure 1.15: As Figure 1.14, but with order reversed.

Algorithm 1.3.37 (choosing the starting node for RCM algorithm)**.** Each variable in the final level set S_k should be tried as a starting node. If one of these nodes produces more than k level sets, then this variable replaces the starting node and the new level sets are examined, continuing in this way until no further increase in the number of level sets is obtained.

Remark 1.3.14. There are two limitations of the methods of this section:

1. They tend to be less effective when the problems are fully two-dimensional, exemplified by a large square grid.

2. The banded methods are the fundamental sequential nature of these orderings. Each computation in the factorization and the solve comes from the previous computation. Thus, the methods tend to do poorly in a parallel environment.

1.3.5 Orderings based on dissection

Definition 1.3.38. There are two desirable form: *bordered block diagonal form* and *doubly-bordered block diagonal form*, which are illustrated in Figure 1.16 and Figure 1.17.

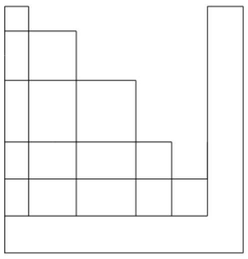


Figure 1.16: Bordered block triangular form.

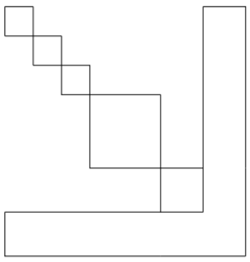


Figure 1.17: Doubly-bordered block triangular form.

Example 1.3.39. To establish the basic idea, consider the graph in Figure 1.18 with the columns in the order $(1, 2, 4, 5, 7, 8, 10, 11, 3, 6, 9)$. A way to motivate this ordering is to look at the entire graph, and select nodes which, when removed, dissect the graph into smaller pieces. The effect of this ordering can be seen in the matrix pattern in Figure 1.19. We make two observations about this matrix pattern:

1. The ordering leads to a matrix that has a bordered block diagonal form with four diagonal blocks. Fill-in is confined to the diagonal blocks and the borders.

2. Factorization of the first four diagonal blocks can be performed independently, allowing an easy exploitation of parallelism.

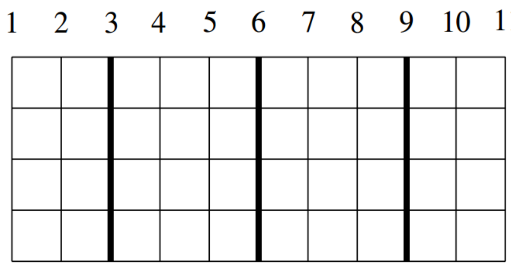


Figure 1.18: A regular 5×11 grid problem, one-way dissected.

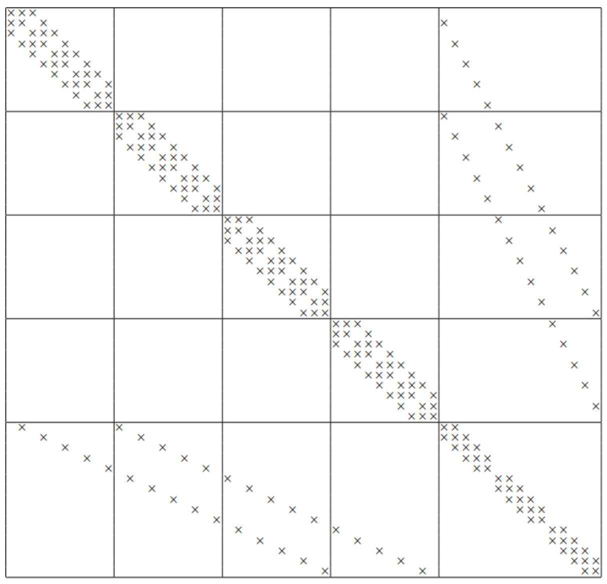


Figure 1.19: The matrix arising from one-way dissection of the problem in Figure 1.18.

Algorithm 1.3.40. Similarly to the way we dealt with the block tridiagonal forms in Section 1.3.4, The factorization of a doubly-bordered block tridiagonal matrix $\mathbf{A}$ can be written in the form

$$\begin{bmatrix} \mathbf{A}_{11} & & & & \mathbf{A}_{1N} \\ & \mathbf{A}_{22} & & & \mathbf{A}_{2N} \\ & & \mathbf{A}_{33} & & \mathbf{A}_{3N} \\ & & & \ddots & \vdots \\ \mathbf{A}_{N1} & \mathbf{A}_{N2} & \mathbf{A}_{N3} & \cdots & \mathbf{A}_{NN} \end{bmatrix} = \mathbf{LU},$$

$\mathbf{L}$ and $\mathbf{U}$ can be written in

$$\mathbf{L} = \begin{bmatrix} \mathbf{I} & & & & \\ & \mathbf{I} & & & \\ & & \mathbf{I} & & \\ & & & \ddots & \\ \mathbf{A}_{N1}\mathbf{D}_1^{-1} & \mathbf{A}_{N2}\mathbf{D}_2^{-1} & \mathbf{A}_{N3}\mathbf{D}_3^{-1} & \cdots & \mathbf{I} \end{bmatrix},$$

$$\mathbf{U} = \begin{bmatrix} \mathbf{D}_1 & & & & \mathbf{A}_{1N} \\ & \mathbf{D}_2 & & & \mathbf{A}_{2N} \\ & & \mathbf{D}_3 & & \mathbf{A}_{3N} \\ & & & \ddots & \vdots \\ & & & & \mathbf{D}_N \end{bmatrix},$$

where

$$\mathbf{D}_i = \mathbf{A}_{ii},\ i = 1, 2, \ldots, N-1. \tag{1.16}$$

$$\mathbf{D}_N = \mathbf{A}_{NN} - \sum_{j=1}^{N-1} \mathbf{A}_{Nj}\mathbf{D}_j^{-1}\mathbf{A}_{jN}. \tag{1.17}$$

To use this form to solve a set of equations $\mathbf{Ax} = \mathbf{b}$, we can carry out the solution in the following way:

$$\mathbf{c}_i = \mathbf{b}_i,\ i = 1, 2, \ldots, N-1 \tag{1.18}$$

$$\mathbf{c}_N = \mathbf{b}_N - \sum_{j=1}^{N-1} \mathbf{A}_{Nj}\mathbf{D}_j^{-1}\mathbf{c}_j, \tag{1.19}$$

followed by the substitution steps

$$\mathbf{x}_N = \mathbf{D}_N^{-1}\mathbf{c}_N, \tag{1.20}$$

$$\mathbf{x}_i = \mathbf{D}_i^{-1}(\mathbf{c}_i - \mathbf{A}_{iN}\mathbf{c}_N),\ i = N-1, \ldots, 1. \tag{1.21}$$

Algorithm 1.3.41 (Finding the dissection cuts for one-way dissection)**.** For automatic one-way dissection of a general problem, George (1980) proposed the algorithm for finding the dissection cuts:

(i) Firstly, generate a good level structure, as in Section 1.3.4.

(ii) Compute the average number of nods at each level, say m.

(iii) Take points from each of the level sets S_j where

$$j = \lfloor i\delta + 0.5 \rfloor,\ i = 1, 2, \ldots \tag{1.22}$$

with spacing

$$\delta = \sqrt{\frac{3m+13}{2}}. \tag{1.23}$$

The formula (1.23) for the spacing was chosen by George on the basis of numerical experiments and analysis of regular grids, with the aim of keeping storage requirements low. It would suffice to place all the points in the level sets (1.22) in the last block and the points in the intervening groups of level sets into the other blocks.

Definition 1.3.42. The *row graph* of a matrix A is the graph with vertices that correspond to the rows of A and with an edge between two vertices if and only if there is at least one column that has an entry in both of the corresponding rows.

Example 1.3.43. There is an example of a matrix and its row graph in Figure 1.20.

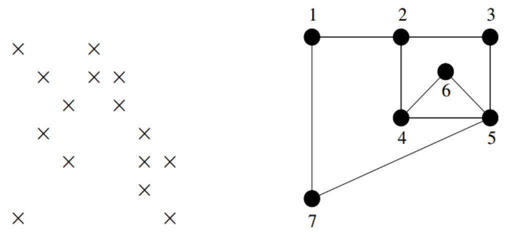


Figure 1.20: A matrix and its associated row graph.

Proposition 1.3.44. The graph of $\mathbf{AA^T}$ is the row graph of $\mathbf{A}$.

Definition 1.3.45. A *hypergraph* is a generalization of a graph in which an edge can join any number of vertices. The generalized edges are called *nets* (also known as *hyperedges*). The vertices in a net are called its *pins* and *the size of a net* is its number of pins. *The degree of a vertex* is the number of nets in which it is a pin. Vertex i can have an associated *weight* w_i and net i can have an associated *cost* c_i.

Definition 1.3.46. In a partition, the set of vertices (rows) is divided into k distinct non-empty *parts* and a net (column) is regarded as being *connected* to a part if it has at least one pin in the part.

Theorem 1.3.47. By permuting the vertices so that those in each part are together and permuting the nets so that those with pins only in part 1 come first, then those with pins only in part 2, and then those with pins only in part k, and finally those with pins in more than one part, we obtain the singly bordered block diagonal form illustrated in Figure 1.21.

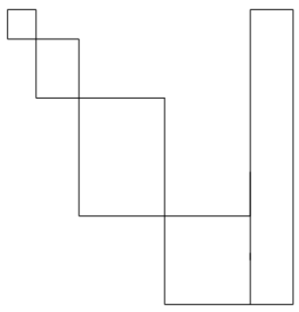


Figure 1.21: A singly bordered block diagonal form.

Definition 1.3.48. The *connectivity* of net (column) i, denoted by λ_i, is defined as the number of parts to which it is connected. It is said to be *cut* (or *external*) if $\lambda_i > 1$. It is the cut nets that we permute to the border.

Definition 1.3.49. A partition has an associated *cost*. The two most common are the sum of the costs c_i of the cut nets and the sum of the weighted sums $c_i(\lambda_i - 1)$ of the cut nets.

Proposition 1.3.50. With $c_i = 1$, the first cost is the number of columns in the border and the second cost corresponds to the amount of communication for distributed matrix-vector multiplication.

Remark 1.3.15. As in the case of graphs, a good partitioning should minimize the cost while maintaining a balance between the parts. If the weight W_i of part i is defined as the sum of the weights w_i of the vertices (rows) in it, a common *balance criterion* is to require

$$W_i \leq W_{avg}(1+e),\ \forall i = 1, \ldots, k \tag{1.24}$$

where W_{avg} is the average weight of a part and e is a tolerance on balance, typically between 0.25 and 1.0.

Remark 1.3.16. We seek to partition the rows into two sets such that only a few columns have entries in both sets.

Definition 1.3.51. The *net-cut* is defined as the number of columns with entries in both partitions and will be equal to the number of border columns in the reordered form.

Definition 1.3.52. A *boundry row* is a row in one set of the current partition that is connected in the row graph to a row in the other set of the partition (has an entry in the same column as a row of the other partition).

Definition 1.3.53. For each row, we can compute the change in the net-cut value caused if the row were moved to the other partition. This is called the *gain*.

Algorithm 1.3.54. The algorithm is used to improve a given partition:

```
Compute the value of the net-cut and load balance for the given partition;
Compute the gain for each boundary row;
Unlock all rows;
do iter = 1, maxit
begin
    while there is an unlocked row and less than maxup consecutive uphill
    moves
    do
        Choose the unlocked boundary row with the highest gain in the
        partition with the greater number of rows. If both partitions are equal,
        arbitrarily choose the row from the first partition. In the case of a tie
        on the gain value, choose the first unlocked row;
        Move this row to the other partition;
        If this is the best partition so far, record it;
        Lock row;
        Update the gains and the list of boundary rows;
    end do;
    Unlock all rows;
    Select best partition found in the previous do loop;
    If net-cut or load balance not improved exit
end;
```

Figure 1.22: A summary of the Kernighan and Lin (1970) algorithm in pseudo-algol.

Remark 1.3.17. To find an appropriate starting point for the Kernighan and Lin algorithm, we would use this algorithm wihin a multilevel scheme. The strategy is coarsening algorithm to coarsen the matrix recursively and then applies the Kernighan and Lin algorithm to the graph of each level.

Algorithm 1.3.55. The strategy used by `MONET` algorithm for coarsening the pattern of the unsymmetric matrix is to recognize and combine pairs of rows that have similar patterns. The algorithm for doing this is given in Figure 1.23.

```
Unlock all rows;
while there are unlocked rows
do
    Choose unlocked row;
    If there exists another unlocked row whose pattern has an entry in common
    with the chosen row then
        Select the row with the most such entries in common;
        Lock the two rows and merge their patterns;
    else
        Lock the row
    endif
end do;
```

Figure 1.23: A summary of the `MONET` coarsening algorithm in pseudo-algol.

This coarsening is continued until the number of rows in the coarse matrix is less than a preset number (default 100 in `HSL_MC66`) or the ratio of the number of rows in consecutive matrices is more than a preset fraction (default 0.75 in `HSL_MC66`).

Remark 1.3.18. The `MONET` approach coarsens the matrix recursively and then applies the Kernighan and Lin algorithm to the graph corresponding to the coarsest matrix from an arbitrary starting partition with roughly equal numbers of rows in its two sets and with its parameters set very high in the hope of getting a near optimal partition on this small matrix.

This partition is then prolongated to the next finer graph through expanding the rows that were collapsed in that level of the coarsening. The Kernighan and Lin algorithm is then applied with this partition as the starting point and the process is repeated up to the finest graph corresponding to the original matrix. The columns of the net-cut are moved to the border.

1.4 Gilbert's LU factorization for sparse matrix

This section is based on Gilbert and Peierls [1988].

Definition 1.4.1 (Symbolic analysis)**.** When solving problems for sparse matrix, it's usually more efficient when working with a fixed datastructure. *Symbolic analysis* is such a precursor to the numerical solution, it includes computations that typically depend only on the nonzero pattern, not the numerical values. This will give a nonzero pattern of the solution, so we can use a fixed datastructure to store it.

Definition 1.4.2. Graph theory is a fundamental tool in sparse matrix techniques. Graph $G_{\mathbf{A}} = (V, E)$ for an $\mathbb{R}^{n\times n}$ sparse matrix $\mathbf{A}$ is defined by vertices $V = \{1, \dots, n\}$, edges $E = \{i \to j : a_{ij} \neq 0\}$ or $E = \{i \to j : a_{ji} \neq 0\}$ which depend on the problem. When the matrix is symmetric, the graph can be undirected.

1.4.1 Solution for sparse triangular systems

Lemma 1.4.3. When we solve $\mathbf{Lx} = \mathbf{b}$, where both $\mathbf{L}$ and $\mathbf{b}$ are sparse and $\mathbf{L}$ is lower triangular, we should first use symbolic analysis to find the nonzero pattern of $\mathbf{x}$. This can be done by a depth-first search in $G_{\mathbf{L}}$ start with $\{i : b_i \neq 0\}$ where $E = \{i \to j : l_{ji} \neq 0\}$.

Proof. Formally, we have the relation

$$x_i = b_i - \sum_{j=1}^{i-1} l_{ij}x_j. \tag{1.25}$$

Notice that we are doing symbolic analysis, so we assume no zero is produced during numerical calculation. So $b_i \neq 0$ or $x_j \neq 0 \cap l_{ij} \neq 0$ means there is at least one nonzero term in (1.25), so

$$\begin{aligned} &b_i \neq 0 \Rightarrow x_i \neq 0, \\ &x_j \neq 0 \cap l_{ij} \neq 0 \Rightarrow x_i \neq 0. \end{aligned}$$

Now, assume $b_i \neq 0$, then $x_i \neq 0$. For all $\{j : l_{ji} \neq 0\}$, we have $x_j \neq 0$, which is equivalent to find all nodes who has an edge start from i. Continue this search so we can find all $x_k \neq 0$ because $b_i \neq 0$, this is indeed a search in $G_{\mathbf{L}}$. Do this for all $\{i : b_i \neq 0\}$, then we find the nonzero pattern of $\mathbf{x}$. Usually, the search is performed by depth-first search. □

Example 1.4.4. For example, when we have $\mathbf{L}$ and $G_{\mathbf{L}}$ as in Figure 1.24.

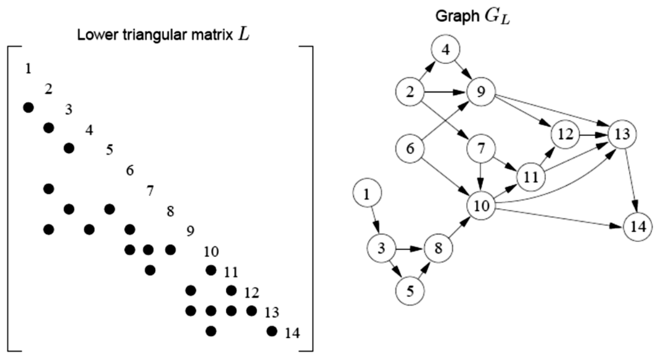


Figure 1.24: $\mathbf{L}$ and corresponding $G_{\mathbf{L}}$

Assuming only $b_4, b_6 \neq 0$, then the depth-first search paths are as in Figure 1.25, where the red points in $\mathbf{L}$ represent the edges in the search paths.

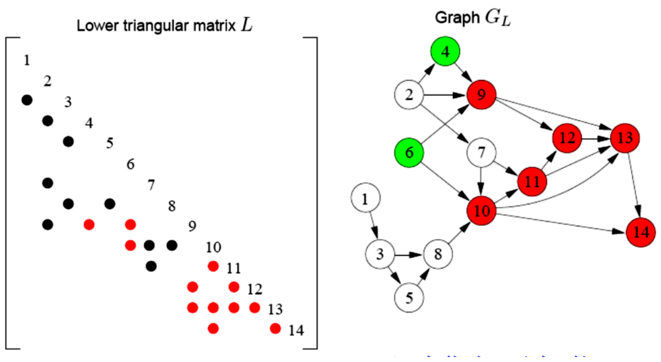


Figure 1.25: The search paths of $\mathbf{Lx} = \mathbf{b}$

Lemma 1.4.5. When we solve $\mathbf{Lx} = \mathbf{b}$ where $\mathbf{L}$ is dense, we use forward substitution, which would solve for the unknowns in increasing order of row number. However, the depth-first search may not find the nonzero positions of x_j in this order, and we don't want to sort it because even a bucket sort would take $\mathcal{O}(n)$ time.

The solution to this problem is adding a stack during the depth-first search, each time the depth-first search backtracks from a vertex, that vertex is pushed into the stack. After all the searches are done, the reverse order of the stack is thus the order which will be used in solving x_j, this order is also called the *topological order*.

Proof. By (1.25), we know that x_i can be calculated as soon as all the $\{x_k : l_{ik} \neq 0 \cap x_k \neq 0\}$ have been calculated. Notice that in the depth-first search, if $l_{ik} \neq 0 \cap x_k \neq 0$, there must be an edge $k \to i$ in the search paths. So, when the depth-first search backtracks from i, all $\{k : l_{ik} \neq 0 \cap x_k \neq 0\}$ are not in the stack yet, otherwise there will be a contradiction. So when we solve x_i in the reverse order of the stack, we will calculate all $\{x_k : l_{ij} \neq 0 \cap x_k \neq 0\}$ before we calculate x_i. □

Theorem 1.4.6. The sparse lower triangular systems $\mathbf{Lx} = \mathbf{b}$ can be solved in $\mathcal{O}(\text{flops}(\mathbf{Lx})+\eta(\mathbf{b}))$ time, where flops($\mathbf{Lx}$) is the number of multiplications of nonzeros when calculating $\mathbf{Lx}$, $\eta(\mathbf{b})$ is the number of nonzeros in $\mathbf{b}$.

Proof. There are two stages in solution, the depth-first search and numerical calculation. The depth-first search takes time proportional to the number of starting vertices plus the number of edges traversed. The number of starting vertices is obviously $\eta(\mathbf{b})$. The number of edges traversed is the number of $\{(i, j) : x_i \neq 0 \cap l_{ji} \neq 0\}$, notice that $(\mathbf{Lx})_j = \sum_{\{i:x_i\neq 0\cap l_{ji}\neq 0\}} l_{ji}x_i$, so the number of edges is exactly flops($\mathbf{Lx}$).

As for the numerical calculation stage, since we know the nonzero structure of $\mathbf{x}$ before we start, we need only initialize and manipulate the positions in the dense vector that corresponding to nonzero positions, thus the whole thing still takes only $\mathcal{O}(\text{flops}(\mathbf{Lx}) + \eta(\mathbf{b}))$ time. □

1.4.2 LU factorization with partial pivoting for sparse matrix

Algorithm 1.4.7. Since partial pivoting is used, we can't predict the nonzero structure of the whole factors. Gilbert and Peierls [1988] then compute the factors column by column just like the left-looking LU factorization. Additionally, the computation of each column of the factors is breaked into a symbolic and numerical stage. Recall the procedure of left-looking LU factorization with partial pivoting.

Algorithm 5: Left-looking LU factorization with partial pivoting

```
Input: $\mathbf{A} \in \mathbb{R}^{n\times n}$
Preconditions : $\mathbf{A}$ is nonsingular
Output: $\mathbf{L}, \mathbf{U} \in \mathbb{R}^{n\times n}$
Postconditions: $\mathbf{L}$ is unit lower
                triangular, $\mathbf{U}$ is
                upper triangular
1 $\mathbf{b}(1 : n) = 0$;
2 for $j = 1 : n$ do
3     Solve $\mathbf{L}(1 : (j-1), 1 : (j-1))\mathbf{U}(1 : (j-1), j) = \mathbf{A}(1 : (j-1), j)$;
4     $\mathbf{b}(j : n) = \mathbf{A}(j : n, j) - \mathbf{L}(j : n, 1 : (j-1))\mathbf{U}(1 : (j-1), j)$;
5     Pivot: swap $b_j$ with the largest-magnitude element of $\mathbf{b}(j : n)$, swap $\mathbf{L}$ and $\mathbf{A}$;
6     $u_{jj} = b_j$;
7     $\mathbf{L}(j : n, j) = \mathbf{b}(j : n)/u_{jj}$;
8 end
```

The left-looking LU factorization with partial pivoting for sparse matrix is actually in the same form, excluding the way to solve $\mathbf{L}(1 : (j-1), 1 : (j-1))\mathbf{U}(1 : (j-1), j) = \mathbf{A}(1 : (j-1), j)$. For sparse matrix, this lower triangular systems can be solved just like what we have discussed in Section 1.4.1.

Theorem 1.4.8. The entire algorithm 1.4.7 can be implmented to run in $\mathcal{O}(\text{flops}(\mathbf{LU}) + \eta(\mathbf{A}))$.

Proof. Define $m = \eta(\mathbf{A})$, $m^* = \eta(\mathbf{L}-\mathbf{I}+\mathbf{U})$, then one can show that $m^* - m \leq \text{flops}(\mathbf{LU})$ because any created nonzero element in $\mathbf{L} - \mathbf{I} + \mathbf{U}$ is used in $\mathbf{LU}$ to balance other terms to make $\mathbf{LU} = \mathbf{A}$ hold.

By Theorem 1.4.6, step 2 and step 3 in algorithm 1.4.7 take total time $\mathcal{O}(\text{flops}(\mathbf{LU}) + m^*)$ plus $\mathcal{O}(n)$ to initialize the mark array for the depth-first search. Step 4 and step 6 each examine every nonzero in $\mathbf{L}$ once, so they take time $\mathcal{O}(m^*)$ over all. Step 5 takes $\mathcal{O}(n)$ time overall. Since $n \leq m \leq m^* \leq \text{flops}(\mathbf{LU}) + m$, the total is $\mathcal{O}(\text{flops}(\mathbf{LU}) + \eta(\mathbf{A}))$. □

1.5 Frontal method

1.5.1 Introduction

Example 1.5.1. In the methods for factorization discussed earlier in this chapter, we have assumed that the matrix $\mathbf{A}$ is given. However, the matrix is 'assembled' as the sum of submatrices sometimes, mostly in finite-element problems.

In a finite-element problem the matrix is a sum

$$\mathbf{A} = \sum_l \mathbf{A}^{[l]}, \tag{1.26}$$

where each $\mathbf{A}^{[l]}$ has entries only in the principal submatrix corresponding to the variables in element l and represents the contributions from this element. It is normal to hold each $\mathbf{A}^{[l]}$ in packed form as a small full matrix together with a list of indices of the variables that are associated with element l.

Definition 1.5.2 (Fully summed)**.** The formation of the sum (1.26) is called *assembly* and involves the elementary operation

$$a_{ij} := a_{ij} + a_{ij}^{[l]}. \tag{1.27}$$

We have used the Algol symbol ':=' here to avoid confusion with the superscript notation $a_{ij}^{(k)}$. We call an entry *fully summed* when all contributions of the form (1.27) have been summed.

Lemma 1.5.3. It is evident that basic operation of Gaussian elimination

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - a_{ik}^{(k)}(a_{kk}^{(k)})^{-1}a_{kj}^{(k)} \tag{1.28}$$

may be performed before all the assemblies (1.27) are complete, provided only the terms in the triple product in (1.28) are fully summed. Each variable can be eliminated as soon as its row and column is fully summed, that is after its last occurrence in a matrix $\mathbf{A}^{[l]}$.

Definition 1.5.4. By Lemma 1.5.3, the elimination operations will be confined to the submatrix of rows and columns corresponding to variables that have not yet been eliminated, but are involved in one or more of the elements that have been assembled. This permits all intermediate working to be performed in a full matrix whose size increases when a variable appears for the first time and decreases when one is eliminated. The pivotal order is determined from the order of the assembly. If the elements are ordered systematically from one end of the region to the other, the active variables form a front that moves along it. For this reason, the full matrix in which all arithmetic is performed is called the *frontal matrix* and the technique is called the *frontal method*.

Example 1.5.5. In Figure 1.26, we show the situation after a set of elimination and just prior to another assembly. The fully-summed rows and columns in blocks $(1,1)$, $(1,2)$ and $(2,1)$ contain the corresponding rows and columns of $\mathbf{L}$ and $\mathbf{U}$. They will not be needed until the solve stage, so may be stored on auxiliary storage as packed vectors. Blocks $(3,1)$ and $(1,3)$ are zero because the eliminated variables are fully summed. Block $(2,2)$ is the frontal matrix, normally held in memory. Blocks $(2,3)$, $(3,2)$, and $(3,3)$ as yet have no contributions, so require no storage.

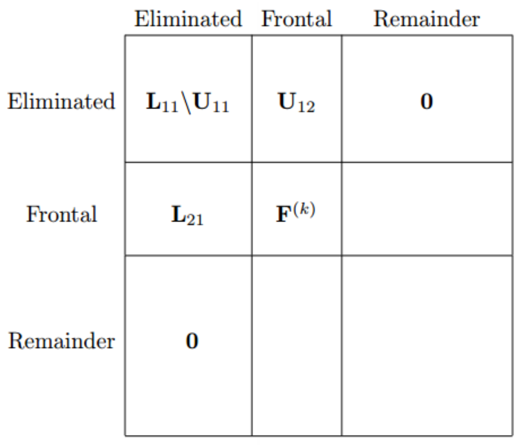


Figure 1.26: Matrix of partially processed frontal method

1.5.2 Element assembly tree

In this section, we assume that the assembled matrix is SPD so that numerical pivoting is not needed for stability.

Definition 1.5.6. Consider the finite-element case. The grouping of elements into substructures and of substructures into bigger substructures until the whole problem is obtained can be expressed as a tree that is know as an *element assembly tree.* In an element assembly tree, each node corresponds to an assembly of a frontal matrix (nothing to do at a leaf node) and subsequent eliminations to form the generated element matrix (also known as the 'contribution block').

Example 1.5.7. For the finite-element problem shown on the left of Figure 1.27, the substructure defined by nested dissection can be represented by the tree shown on the right of Figure 1.27

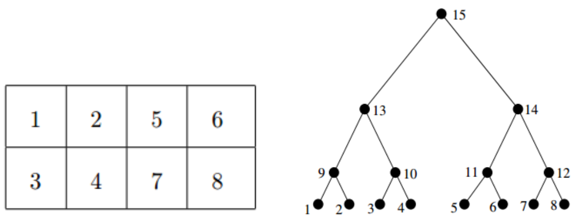


Figure 1.27: A finite-element problem and the corresponding element assembly tree

This nested grouping of elements into substructures can also be represented by bracketing of the sum

$$\mathbf{A} = \sum_{l} \mathbf{A}^{[l]} = ((\mathbf{A}^{[1]} + \mathbf{A}^{[2]}) + (\mathbf{A}^{[3]} + \mathbf{A}^{[4]}) + (\mathbf{A}^{[5]} + \mathbf{A}^{[6]}) + (\mathbf{A}^{[7]} + \mathbf{A}^{[8]})).$$

For the above element assembly tree, we may perform the eliminations as follows:

(i) Eliminate all variables that belong in only one element, storing the resulting pivotal rows and columns, and setting the resulting generated element matrices aside temporarily.

(ii) Assemble the resulting matrices in pairs and eliminate any variable that is internal to a single pair; store the resulting generated element matrices $\mathbf{A}^{[9]} \sim \mathbf{A}^{[12]}$.

(iii) Repeat the second step until there is only one structure.

1.5.3 Elimination tree

In this section, we take as our starting point a given SPD matrix of order n.

Definition 1.5.8. Just like the element assembly tree, we can construct a tree called *elimination tree* to arrange the ordering of the elimination. It has a node for each row of the matrix, if node a is a child of node b, it means that the elimination of pivot a will have contribution to b.

Lemma 1.5.9. Suppose row k has the first off-diagonal entry in the column $l > k$, then row k must appear ahead of row l since otherwise the numerical values in row l when it becomes pivotal will be different. We represent this by an edge (k, l). For any other entry $m > l$, the elimination at step k will fill entry $a_{lm}^{(k)}$, if not already filled, so m must be an ancestor of l (when m is smaller than the first off-diagonal entry in row l when eliminating pivot l, m will be the parent of l). By this procedure, for any node k_1, we can get a sequence of edges to nodes $k_1 < k_2 < ... < k_r$, ending at a node k_r corresponding to a row that has no entries to the right of the diagonal when eliminating it.

Example 1.5.10. In Figure 1.28, row 1 and 2 may be interchanged without having any effect on the operations performed, but row 1 must precede row 3. Since 3 is the first off-diagonal entry when eliminating pivot 1, it will be the parent of 1. Since $a_{24}, a_{25} \neq 0$, there will be a fill-in in position $a_{45}^{(2)}$, so row 4 must precede row 5. Since 5 is the first off-diagonal entry when eliminating pivot 4, it will be the parent of 4. The corresponding elimination tree is shown on the right of Figure 1.28.

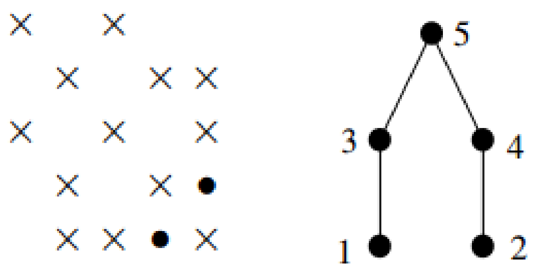


Figure 1.28: A matrix (each fill-in is marked as a point) and the corresponding elimination tree

Algorithm 1.5.11 (Generate the elimination tree)**.** **?** has given an efficient way to generate the elimination tree, essentially in time proportional to the number of entries in the original matrix.

We first consider what happens if we try to generate the elimination tree for a symmetric matrix that is reducible, that is, a permutation of a block diagonal matrix. The equations for a single block are then independent of the other equations and the elimination tree for this block will have as its root the node corresponding to the variable in the block that is last in the pivotal order. If there are m blocks, there will be m such elimination trees, so we have a forest. So now we only consider irreducible matrices.

The k-th stage $(k = 1, ...n)$ of Liu's algorithm starts with the elimination forest for the leading submatrix of order $k-1$, this will often be a forest rather than a tree since a submatrix of an irreducible matrix may be reducible. He then add the entries of column k of the original matrix one by one and the forest is updated to correspond. He starts by adding a one node for column k. When here comes an adding entry a_{ik}, $i<k$, notice that k is the largest node for now, we will find that k is the parent of i's root by Lemma 1.5.9. Follow this procedure until all the nonzeros in column k of the leading submatrix of order k have been considered, the forest is fully updated and we are ready to go to stage $k+1$.

Example 1.5.12. We show how algorithm 1.5.11 work in Figure 1.29, which corresponding to updating the forest during the processing of the final column of Figure 1.28 . When we process a_{25} we find that node 2 has node 4 as its root, so there is a fill-in at position $(5,4)$ so node 5 is moved to be the parent of node 4. When we process a_{35}, we find that node 3 is already a root, so node 3 is given node 5 as its parent.

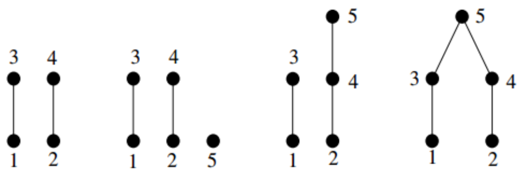


Figure 1.29: Updating the forest during the processing of the final column of Figure 1.28

Lemma 1.5.13. Having constructed the elimination tree, we may use it to factorize the matrix, or another matrix with the same soarsity pattern. We visit the nodes in some topological order (basicly from leaves to root). At each node, we add together the original 'elements' (nonzeros in the pivot's row and column) and the generated element matrices from the children (if any), perform the elimination operations with the packed matrix, store the calculated rows of $\mathbf{U}$, and hold the generated element matrix for use later.

Example 1.5.14. Actually, when no pivoting is needed, frontal method can be easily generalized to matrices only have symmetric pattern. For example, a matrix like

$$\mathbf{A}=\begin{bmatrix}2&0&0&2&1\\0&1&-1&0&0\\0&-1&0&-2&0\\4&0&2&14&0\\-6&0&0&0&-2\end{bmatrix}.$$

When eliminating pivot 1, we consider the first row and column, which is

$$\mathbf{A}_1^{(145)}=\begin{bmatrix}2&2&1\\4&0&0\\-6&0&0\end{bmatrix},$$

where (145) represent the nonzero indices. The eliminated matrix will be

$$\mathbf{F}_1^{(145)}=\begin{bmatrix}2&2&1\\2&-4&-2\\-3&6&3\end{bmatrix}.$$

The generated element matrix (contribution block) is

$$\mathbf{CB}_1^{(45)}=\begin{bmatrix}-4&-2\\6&3\end{bmatrix},$$

which is also called the *Schur complement.* (45) means the generated elements will have contribution when eliminating pivots 4 and 5.

Similarly, when eliminating pivot 2,

$$\mathbf{A}_2^{(23)}=\begin{bmatrix}1&-1\\-1&0\end{bmatrix},\ \mathbf{F}_2^{(23)}=\begin{bmatrix}1&1\\-1&-1\end{bmatrix},$$
$$\mathbf{CB}_2^{(3)}=\begin{bmatrix}-1\end{bmatrix}.$$

When eliminating pivot 3, notice that there is a generated element matrix $CB_2^{(3)}$, we add together the original elements and the generated element matrix and get

$$\mathbf{A}_3^{(34)}=\begin{bmatrix}-1&-2\\2&0\end{bmatrix},\ \mathbf{F}_3^{(34)}=\begin{bmatrix}-1&-2\\-2&-4\end{bmatrix},$$
$$\mathbf{CB}_3^{(4)}=\begin{bmatrix}-4\end{bmatrix}.$$

Similarly, when eliminating pivot 4, there are generated element matrices $CB_1^{(45)}$ and $CB_3^{(4)}$, so

$$\mathbf{A}_4^{(45)}=\begin{bmatrix}6&-2\\6&3\end{bmatrix},\ \mathbf{F}_4^{(45)}=\begin{bmatrix}6&-2\\1&5\end{bmatrix},\ \mathbf{CB}_4^{(5)}=\begin{bmatrix}5\end{bmatrix}.$$

Finally, $\mathbf{A}_5^{(5)}=3=\mathbf{F}_5^{(5)}$ and the factorization is complete. The elimination tree is as in Figure 1.30.

$$\begin{tikzcd}
 & F_5^{(5)} & \\
 & F_4^{(45)} \arrow[u, "CB_4^{(5)}"'] \arrow[dr, dash, "CB_3^{(4)}"] & \\
F_1^{(145)} \arrow[ur, "CB_1^{(45)}"] & & F_3^{(34)} \\
 & & F_2^{(23)} \arrow[u, "CB_2^{(3)}"']
\end{tikzcd}$$

Figure 1.30: Elimination tree

1.5.4 Numerical pivoting

Algorithm 1.5.15. The frontal methods can also be applied to systems which need numerical pivoting. As shown in Figure 1.31, where the block of the front that is partially summed is labelled *ps* and the parts that are fully summed are labelled fs. Any entry whose row and column is fully summed (that is, any entry in block $(1,1)$ of Figure 1.31) may be used as a pivot. We can use any pivoting strategy including partial pivoting, threshold pivoting and so on. Notice that if unsymmetric interchanges are included, the index list for the rows in the front will differ from columns and will require separate storage.

$$\begin{array}{c|cc|c|}
 & \multicolumn{2}{c}{\text{Front}} & \multicolumn{1}{c}{\text{Remainder}} \\ \hline
\text{Front} & fs & fs & \mathbf{0} \\ \cline{2-4}
 & fs & ps & \\ \hline
\text{Remainder} & \mathbf{0} & & \\ \hline
\end{array}$$

Figure 1.31: Reduced submatrix after an assembly and prior to another set of eliminations

It is possible that we cannot choose pivots for the whole of the frontal matrix because of small entries in the front or large off-diagonal entries that lie outside the block of the fully-summed rows and columns (that is, because they lie in block $(1,2)$ or block $(2,1)$ of Figure 1.31). In this case, we simply leave the variables in the front, continue with the next assembly, and then try again. It is always possible to complete the factorization using this pivotal strategy because each entry eventually lies in a fully-summed row and in a fully-summed column.

Bibliography

L. Foster. The growth factor and efficiency of gaussian elimination with rook pivoting. *Journal of Computational and Applied Mathematics*, 98:177–194, 11 1997. doi: 10.1016/S0377-0427(97)00154-4.

J. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *Siam Journal on Scientific and Statistical Computing*, 9, 09 1988. doi: 10.1137/0909058.

J. Reid. A note on the stability of gaussian elimination. *IMA Journal of Applied Mathematics*, 8, 12 1971. doi: 10.1093/imamat/8.3.374.

D. Ruiz and B. Uçar. A symmetry preserving algorithm for matrix scaling. *SIAM Journal on Matrix Analysis and Applications*, 35, 07 2014. doi: 10.1137/110825753.

J. Wilkinson. Error analysis of direct methods of matrix inversion. *J. ACM*, 8:281–330, 07 1961. doi: 10.1145/321075.321076.